ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ
ΤΜΗΜΑ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Mechanisms for Monitoring Optimization in Cloud Computing Environments

Μηχανισμοί Βελτιστοποίησης εποπτείας σε περιβάλλοντα υπολογιστικών νεφών

JEAN-DIDIER TOTOW TOM-ATA

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Δρ. Dimosthenis Kiriazis

Piraeus, February 2020

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ
ΤΜΗΜΑ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μηχανισμοί Βελτιστοποίησης εποπτείας σε περιβάλλοντα υπολογιστικών νεφών

Optimization mechanism of a monitoring in cloud computing environment

JEAN-DIDIER TOTOW TOM-ATA

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Δρ. Dimosthenis Kiriazis

Piraeus, February 2020

Abstract

In big data environment that delivers a complete pioneering stack, based on a frontrunner infrastructure management system that drives decisions according to data aspects, thus being fully scalable, runtime adaptable and high-performant to address the emerging needs of big data operations and data-intensive applications, the data-driven platform should collect and analyse/evaluate periodically metrics from different components involved in a specific application performance. We distinguish three groups of components involved in the environment performance: the infrastructure where applications are running (Kubernetes[1], openshift[2] etc …), data components (object storing systems, databases) and applications running. These components generate a huge amount of metrics which have to be collected, evaluated (quality of service) stored and exposed to a decision component in real-time and an ad-hoc mode.

Metric could be memory usage, a cpu consumption, number of processes, application starting time etc. We can clearly understand that some metrics could be produced by a batch job and some others are produced periodically so the need of providing different mechanisms to collect and consume them.

Building a monitoring engine implementing functionalities listed above introduces a considerable delay from the moment a metric is collected and the moment this metric is available for consumption due to all processing units in between. The bigger is the amount of measurements, the more information the platform can receive and better will be the decision. However, the amount of data is directly proportional to the delay related earlier.

This delay affects the performance of the decision component since this last should catch events as soon as possible. In order to enable later analysis on metrics, the monitoring engine should provide methods for storing metrics. However, measurements are taken periodically from applications for being used for analysis and historical purpose.

Table of content

I. Introduction

I.1 Problem description

This project consists of developing a monitoring engine for big data environment. In other words, we are building a monitoring engine of platform which handle huge amount data having different format (type of content) and where those information are arriving at high speed. This initiative comes from a real challenges that have to be addressed in a big data environment.
In this part, we will be illustrating different scenario from which the initiative is based on. The business usage scenarios and initial requirements elicited from each of the three business use cases of the BigDataStack project.These requirements should be considered as Stakeholder Requirements focused on specific solutions as required by specific User Enterprises. The business scenarios are representative of a significant business need or problem, and enables data, technology and service providers to understand the value to the customer organization of a developed Big Data solution. Each scenario describes the different usage from a use case perspective at a high-level description. It is not the intention to define the complete and detailed scenarios needed for the development of the solution, rather that the descriptions are more related with defining the behaviour and the scope to identify the necessities and align the architecture definition with the uses case from the beginning. Moreover, the scenarios are by no means complete, as the project has two additional iterations to upgrade and refine them, however, they provide an overview on the main behavioural patterns involving the different and aims to define and align the initial design of the architecture. Scenario descriptions are complemented with UML Use Case Diagrams to identify the different actors, prerequisites and the description of the behaviour.Each use case can identify one or more scenarios depending on the complexity or the scope of the definition. For instance, on one side,the necessity for the analysis of the data services and data-intensiveness of the provision (at the dimensioning phase), and on the other side, the scenario for the operational phase where the defined Quality of Service (QoS) and rules should be applied. Thus, this can be described only in one scenario (more complex) or can be split into two scenarios differentiating clearly the objectives, the behaviour and the actors. It should be the decision of each use case provider to take the approach that best suits their purpose.

Use case 1 real time ship management

Fig.

This scenario addresses two main challenges:

- Maintenance prediction: This challenge consists of creating an environment where data from different ship sensors will be gathered and used for predicting the potential components of the ship that will require maintenance. This feature is crucial in the business perspective since it enables better action planning, minimize reparation cost. Ship engines and other relevant machinery need to achieve high availability not only to deliver transport services (and thus ensure availability of resources) but also for operational safety, occupational health and environmental impact purposes. High availability of ship engines and machines can only be achieved if they are kept under proper conditions using applicable maintenance strategies, thus the monitoring of machinery has become even more critical to meet the maintenance requirements and achieve predictive maintenance. The latter is based on data that are exploited to estimate the type of failure and time to failure.

- Dynamic routing: Once a malfunction is identified and the technical department is informed (Fleet manager, coordinator), spare parts or actions to be taken for maintenance should be clarified from the technical department to the supplies department. The supply department should order the required spare part and proceed with the requisition and delivery process of the part to the vessel. The cost of the spare part depends on the location of the vessel, on the distance where the closest port is, and on the supplier, while some qualitative criteria must be taken into account. Usually, each shipping company has a list of suppliers who are trusted. Thus, the supply department wishes to minimize the cost of the ordered spare part without compromising the quality of the part itself and replace it on time without letting the damage on the main engine put the vessel off-hire.

Use case 2: Connecting customers

Fig

This scenario refers to the use case of Connected Consumer: Multi-sided market ecosystem. We will be describing the scenario by providing detailed information regarding the requirement.

In today’s world where information is accessed instantly and competition is just as fast as one click away, attracting and keeping customers is crucial for survival of of a business. Thus , Predictive analysis is the challenge. It can help predict which consumers are the most loyal or which potential buyers are more likely to purchase a certain product or service, opening new opportunities for retailers, providing new business prospects to customers, with improved shopping experience for consumers and new business opportunities for traders.

In this business domain, Eroski, one of the largest distribution companies in Spain with more than 35.000 workers, is collaborating with ATOS in the definition and test of a use-case related to the grocery business. It is also contributing with real data for the development of the project. The goal of this scenario is to provide data insights to EROSKI to better understand how to create and offer added-value services to their consumers. In this context, the use case objective is to predict both which products and which promotions are more likely to be interesting for the customers at the right time. In this way, EROSKI can adapt the most appropriate message for each customer and send it at the right time and through the most appropriate channel, thus increasing the ROI of their marketing activities.

From the analysis of different data sources provided by Eroski, the goal is first to predict the list of products that customers with recurrent purchases will need in the current purchase period (trend). Afterwards, add to this prediction those products that can be interesting for the user based on other similar user's behaviour (cross-selling). Finally, thanks to a deep knowledge of the customer profile, the goal is also to incorporate those promotions that can be interesting for each customer.

Additionally, a scenario that describes a demonstrator that will help users to display and test recommendations made by the user has also been included.

I.2 Approach

Based on the two use cases related above, we can find common points in order to define the requirements of the platform that has to be built for allowing the implementation of these two use case. This process will lead us to the overall understanding of the platform. We can clearly understand that the platform must provide a capability of collecting information of different type, format and coming at a very high speed. The platform must be able to handle a huge dataset, process them and perform some machine learning on them. thus , the environment must implement quick and efficient mechanism of storing and big data analysis. Applications deployed to this platform handle a non constant load. We should be able to implement elasticity capability where resources can be dynamically allocated and deallocated.
Those main requirements lead to build a monitoring mechanism of the platform that will provide the platform in real-time the current status and itself and of all the applications running allowing the reactivity of the platform. We also need a monitoring mechanism that can enable time series prediction for the proactivity of the platform. This functionality will prevent the platform of unnecessary adaptation.

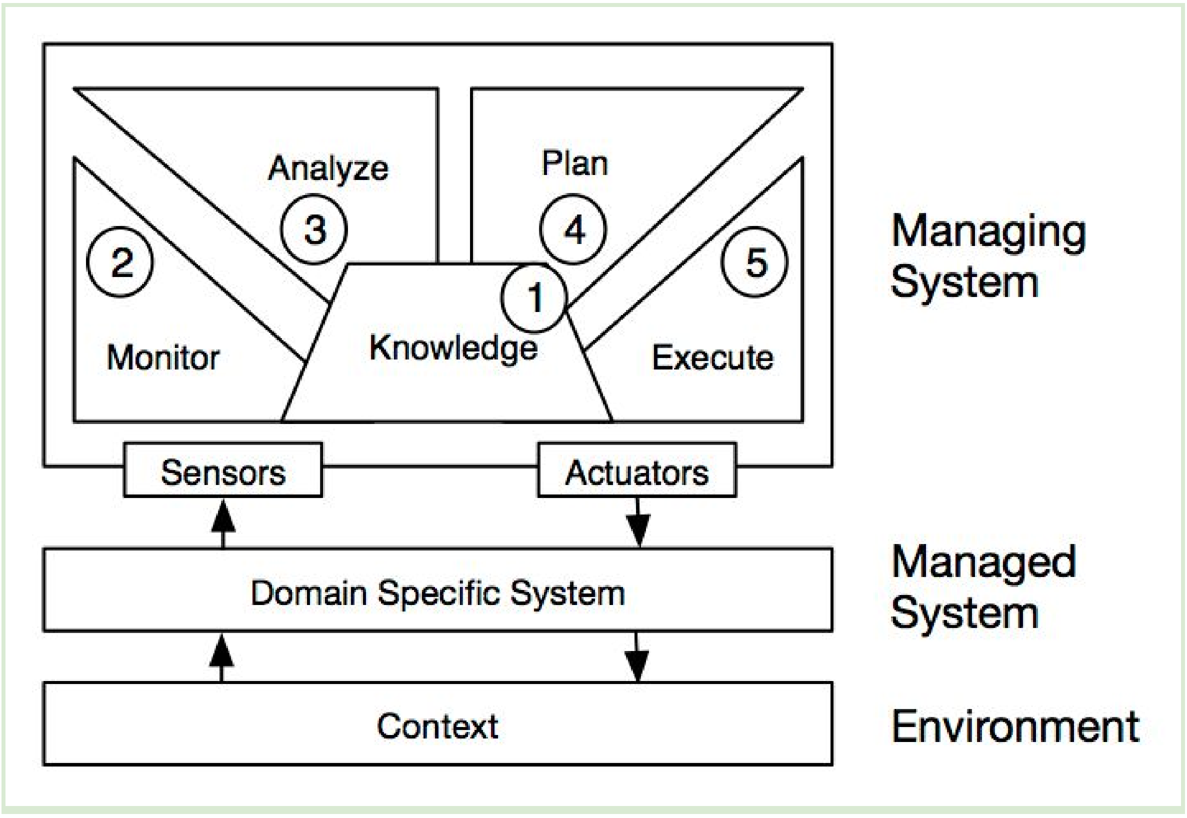


Fig.

The previous image describes the functional design of the platform. From this diagram we will focus our attention on the managing system. The later is composed of the following units:

- Sensors: agents from which measurements are taken. They are reading measurement periodically and exposing them for consumption by the managing system.
- Actuators: those are elements which command the platform to take a specific action.
- Monitoring: This unit is connected to sensors in order to collect metrics, aggregate them if needed, then prepare them for consumption.
- Analyse: This is the intelligent part of the platform, this unit compares measurement with the model which defines the objective.
- Knowledge: This component contains models of the desired application
- Plan: This unit defines different steps and their execution order for applying modifications desired
- Execution: In this components, modifications will be transformed to commands

I.3 System monitoring

A system monitoring is a method consisting of visualizing resources and system performance. System monitoring is commonly used to keep of the system performance. It has the capability of tracking the CPU activity, memory or space disk used, network activities such as bandwidth, the number of packets received, the number of packets lost etc…

Monitoring engine is a crucial element in data-driven environment since decisions are based on the collected system performance indicators or metrics. A monitoring engine for data-driven platform should implement the capability of collected an enormous amount of data and handle them quickly so that to available for decision making elements.
The general architecture of a monitoring system is composed of two parts : The agent or exporter which implements functions for reading metrics and a manager which is a collector. Most of the time the manager asks for metrics by sending a "get" request specifying the name of the metric then the agent replies by sending back the corresponding metric's value and some information related such as : time, instance that generates the metric, labels etc. There is a possibility the agent/exporter to start the communication, therefore we talk about **pushing** mode which needs a streaming channel or persistent connection. In a big environment where many applications produce metrics per second, there is a need to group metrics by producer(component that generates metrics) then expose them together. It is very important to determine the interval of time (scrape time) where measurements will be read. If this time is very big, the platform may lose some important events. In case this interval is very small, the platform could be overloaded with useless or meaningless duplicated metric's value. Therefore it's imperative the component producer owner to determine the correct scrape interval.

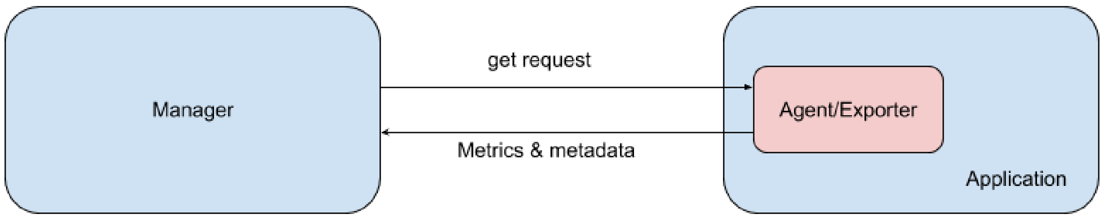


Fig. 1.1

I.4 Collection methods

Metrics collection consists of gathering measurements from applications. The techniques used depend on the type of the application, the monitoring collector capabilities and also the use case. The monitoring collector used in the context of this final project is Prometheus.
Prometheus is an open source application used for event monitoring and alerting. It records real-time metrics in a time series database (allowing for high dimensionality) built using a HTTP pull model, with flexible queries and real-time alerting. The project is written in Go and licensed under the Apache 2 License, with source code available on GitHub, and is a graduated project of the Cloud Native Computing Foundation, along with Kubernetes and Envoy.

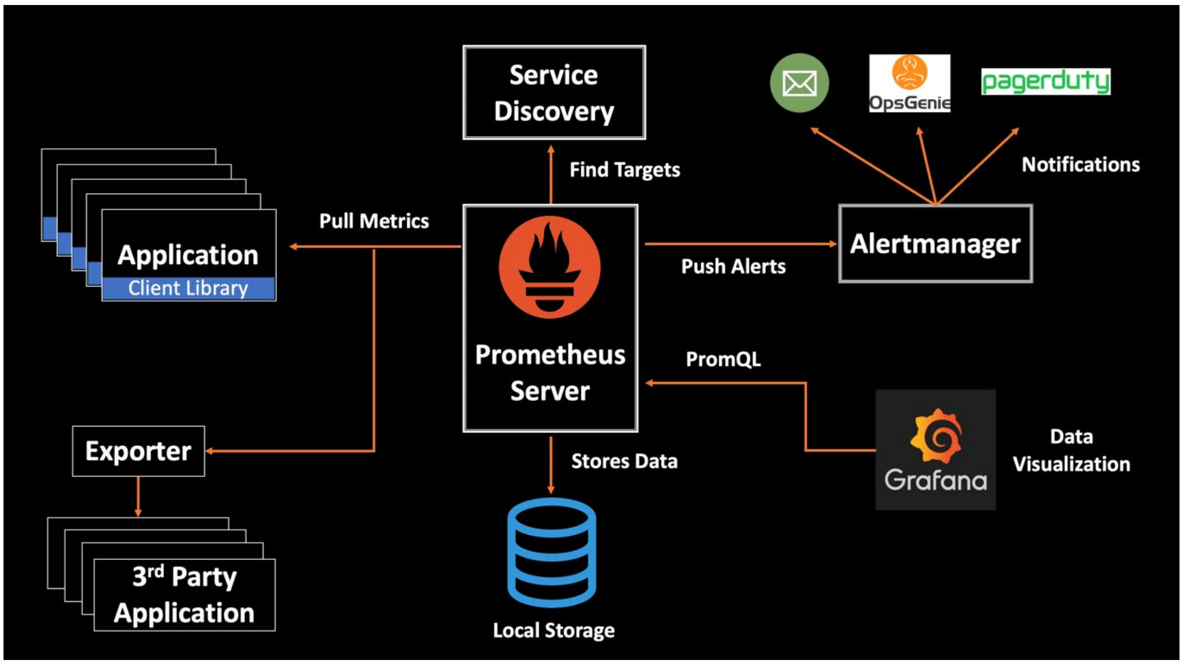


Fig.

Prometheus collects data in the form of time series. The time series are built through a pull model: the Prometheus server queries a list of data sources (exporters) at a specific polling frequency determined by the scraping time in the configuration of the collector. Each of the data sources serves the current values of the metrics for that data source at the endpoint queried by Prometheus. The Prometheus server then aggregates data across the data sources. Prometheus has a number of mechanisms to automatically discover resources that it should be used as data sources.
We can distinguish 6 main parts from Prometheus' internal architecture:

- Pull metrics: This is the entry where metrics are collected. The part uses an http oriented connexion bringing a smooth and standard method for collecting metrics from application implemented prometheus' client and from exporters.

- Service discovery: This is the mechanism by which Prometheus can discover new metrics source. This mechanism supports file, dns and some embedded method for discovering new sources from which metrics can be gathered. This method allows the implementation of automated metrics source configuration.
  Here are the support services:

  Azure virtual machines: **azure_sd_configs**

The following are the available labels:

__*meta_azure_machine_id*: the machine ID

__*meta_azure_machine_location*: the location the machine runs in

__*meta_azure_machine_name*: the machine name

__*meta_azure_machine_os_type*: the machine operating system

__*meta_azure_machine_private_ip*: the machine's private IP

__*meta_azure_machine_public_ip*: the machine's public IP if it exists

__*meta_azure_machine_resource_group*: the machine's resource group

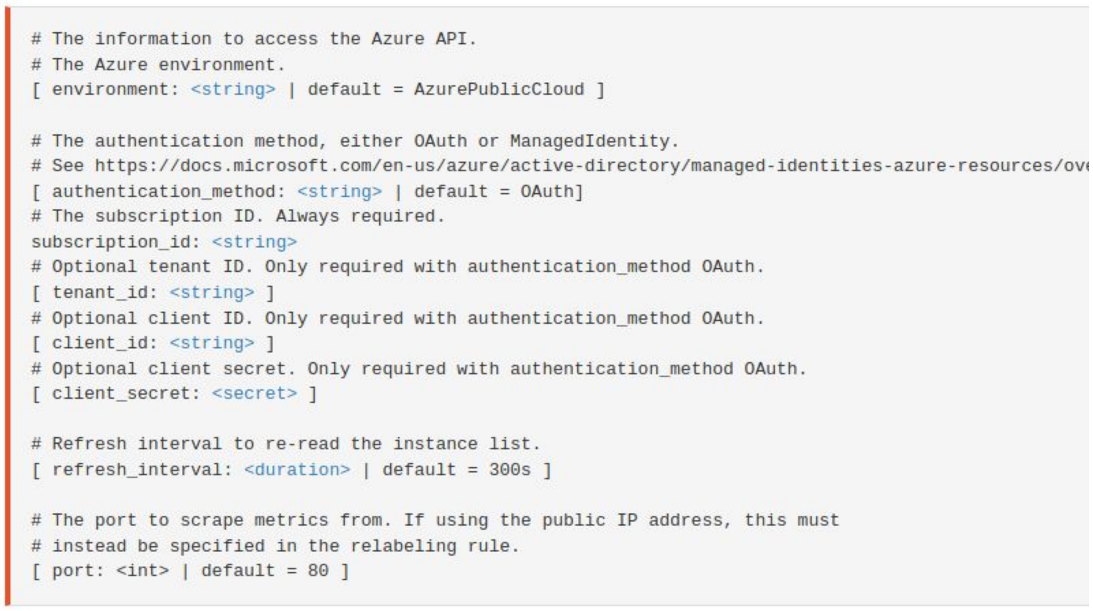
__*meta_azure_machine_tag_<tagname>*: each tag value of the machine

__*meta_azure_machine_scale_set*: the name of the scale set which the vm is part of (this value is only set if you are using a scale set)

__*meta_azure_subscription_id*: the subscription ID

__*meta_azure_tenant_id*: the tenant ID

And the configuration is the follow:



```
# The information to access the Azure API.
# The Azure environment.
[ environment: <string> | default = AzurePublicCloud ]

# The authentication method, either OAuth or ManagedIdentity.
# See https://docs.microsoft.com/en-us/azure/active-directory/managed-identities-azure-resources/ove
[ authentication_method: <string> | default = OAuth]
# The subscription ID. Always required.
subscription_id: <string>
# Optional tenant ID. Only required with authentication_method OAuth.
[ tenant_id: <string> ]
# Optional client ID. Only required with authentication_method OAuth.
[ client_id: <string> ]
# Optional client secret. Only required with authentication_method OAuth.
[ client_secret: <secret> ]

# Refresh interval to re-read the instance list.
[ refresh_interval: <duration> | default = 300s ]

# The port to scrape metrics from. If using the public IP address, this must
# instead be specified in the relabeling rule.
[ port: <int> | default = 80 ]
```

Fig. Azure

Consul platform: consul_sd_config
Those are the available labels:

__*meta_consul_address*: the address of the target

__*meta_consul_dc*: the datacenter name for the target

__*meta_consul_tagged_address_<key>*: each node tagged address key value of the target

__*meta_consul_metadata_<key>*: each node metadata key value of the target

__*meta_consul_node*: the node name defined for the target

__*meta_consul_service_address*: the service address of the target

__*meta_consul_service_id*: the service ID of the target

__*meta_consul_service_metadata_<key>*: each service metadata key value of the target

__*meta_consul_service_port*: the service port of the target

__*meta_consul_service*: the name of the service the target belongs to

__*meta_consul_tags*: the list of tags of the target joined by the tag separator

The configuration block is the follow:

```
# The information to access the Consul API. It is to be defined
# as the Consul documentation requires.
[ server: <host> | default = "localhost:8500" ]
[ token: <secret> ]
[ datacenter: <string> ]
[ scheme: <string> | default = "http" ]
[ username: <string> ]
[ password: <secret> ]

tls_config:
  [ <tls_config> ]

# A list of services for which targets are retrieved. If omitted, all services
# are scraped.
services:
  [ - <string> ]

# See https://www.consul.io/api/catalog.html#list-nodes-for-service to know more
# about the possible filters that can be used.

# An optional list of tags used to filter nodes for a given service. Services must contain all tags
tags:
  [ - <string> ]

# Node metadata used to filter nodes for a given service.
[ node_meta:
  [ <name>: <value> ... ] ]

# The string by which Consul tags are joined into the tag label.
[ tag_separator: <string> | default = , ]

# Allow stale Consul results (see https://www.consul.io/api/features/consistency.html). Will reduce
[ allow_stale: <bool> ]

# The time after which the provided names are refreshed.
# On large setup it might be a good idea to increase this value because the catalog will change all
[ refresh_interval: <duration> | default = 30s ]
```

DNS Discovery service <dns_sd_configs> : Domain Name Service prometheus discovery mechanism allows to find different targets by querying periodically.
This sd service has one available label __*meta_dns_name* which is a set of domain record.

The configuration is the follow:

```
# A list of DNS domain names to be queried.
names:
  [ - <domain_name> ]

# The type of DNS query to perform.
[ type: <query_type> | default = 'SRV' ]

# The port number used if the query type is not SRV.
[ port: <number>]

# The time after which the provided names are refreshed.
[ refresh_interval: <duration> | default = 30s ]
```

Prometheus provides a discovery service for AWS Instances, this service allows discovering targets from Amazon instances. <ec2_sd_config>
Here are the available labels for this service.
__*meta_ec2_availability_zone*: the availability zone in which the instance is running

__*meta_ec2_instance_id*: the EC2 instance ID

__*meta_ec2_instance_state*: the state of the EC2 instance

__*meta_ec2_instance_type*: the type of the EC2 instance

__*meta_ec2_owner_id*: the ID of the AWS account that owns the EC2 instance

__*meta_ec2_platform*: the Operating System platform, set to 'windows' on Windows servers, absent otherwise

__*meta_ec2_primary_subnet_id*: the subnet ID of the primary network interface, if available

__*meta_ec2_private_dns_name*: the private DNS name of the instance, if available

__*meta_ec2_private_ip*: the private IP address of the instance, if present

__*meta_ec2_public_dns_name*: the public DNS name of the instance, if available

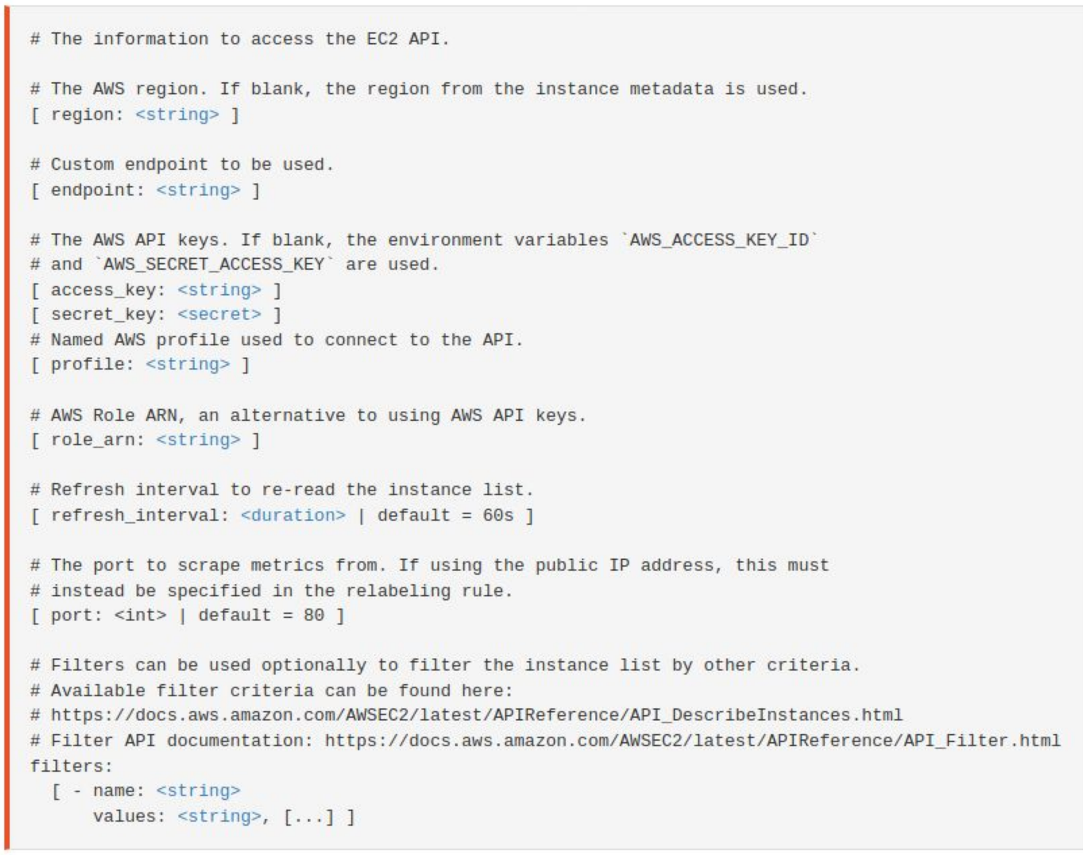
__*meta_ec2_public_ip*: the public IP address of the instance, if available

__*meta_ec2_subnet_id*: comma separated list of subnets IDs in which the instance is running, if available

__*meta_ec2_tag_<tagkey>*: each tag value of the instance

__*meta_ec2_vpc_id*: the ID of the VPC in which the instance is running, if available

The configuration part is the follow:



```
# The information to access the EC2 API.

# The AWS region. If blank, the region from the instance metadata is used.
[ region: <string> ]

# Custom endpoint to be used.
[ endpoint: <string> ]

# The AWS API keys. If blank, the environment variables `AWS_ACCESS_KEY_ID`
# and `AWS_SECRET_ACCESS_KEY` are used.
[ access_key: <string> ]
[ secret_key: <secret> ]
# Named AWS profile used to connect to the API.
[ profile: <string> ]

# AWS Role ARN, an alternative to using AWS API keys.
[ role_arn: <string> ]

# Refresh interval to re-read the instance list.
[ refresh_interval: <duration> | default = 60s ]

# The port to scrape metrics from. If using the public IP address, this must
# instead be specified in the relabeling rule.
[ port: <int> | default = 80 ]

# Filters can be used optionally to filter the instance list by other criteria.
# Available filter criteria can be found here:
# https://docs.aws.amazon.com/AWSEC2/latest/APIReference/API_DescribeInstances.html
# Filter API documentation: https://docs.aws.amazon.com/AWSEC2/latest/APIReference/API_Filter.html
filters:
  [ - name: <string>
      values: <string>, [...] ]
```

OpenStack SD configurations allow retrieving scrape targets from OpenStack Nova instances. <openstack_sd_configs>

This service provides two groups of labels, first are the label related to the hypervisor.

__*meta_openstack_hypervisor_host_ip*: the hypervisor node's IP address.

__*meta_openstack_hypervisor_name*: the hypervisor node's name.

__*meta_openstack_hypervisor_state*: the hypervisor node's state.

__*meta_openstack_hypervisor_status*: the hypervisor node's status.

__*meta_openstack_hypervisor_type*: the hypervisor node's type.

Second are labels related to Nova instances.

__*meta_openstack_address_pool*: the pool of the private IP.

__*meta_openstack_instance_flavor*: the flavor of the OpenStack instance.

__*meta_openstack_instance_id*: the OpenStack instance ID.

__*meta_openstack_instance_name*: the OpenStack instance name.

__*meta_openstack_instance_status*: the status of the OpenStack instance.

__*meta_openstack_private_ip*: the private IP of the OpenStack instance.

__*meta_openstack_project_id*: the project (tenant) owning this instance.

__*meta_openstack_public_ip*: the public IP of the OpenStack instance.

__*meta_openstack_tag_<tagkey>*: each tag value of the instance.

__*meta_openstack_user_id*: the user account owning the tenant.

The follow is the configuration:

```
# The information to access the OpenStack API.

# The OpenStack role of entities that should be discovered.

role: <openstack_role>

# The OpenStack Region.

region: <string>

# identity_endpoint specifies the HTTP endpoint that is required to work with

# the Identity API of the appropriate version. While it's ultimately needed by

# all of the identity services, it will often be populated by a provider-level

# function.

[ identity_endpoint: <string> ]

# username is required if using Identity V2 API. Consult with your provider's

# control panel to discover your account's username. In Identity V3, either

# userid or a combination of username and domain_id or domain_name are needed.

[ username: <string> ]

[ userid: <string> ]

# password for the Identity V2 and V3 APIs. Consult with your provider's

# control panel to discover your account's preferred method of authentication.
```

```
[ password: <secret> ]

# At most one of domain_id and domain_name must be provided if using username

# with Identity V3. Otherwise, either are optional.

[ domain_name: <string> ]

[ domain_id: <string> ]

# The project_id and project_name fields are optional for the Identity V2 API.

# Some providers allow you to specify a project_name instead of the project_id.

# Some require both. Your provider's authentication policies will determine

# how these fields influence authentication.

[ project_name: <string> ]

[ project_id: <string> ]

# The application_credential_id or application_credential_name fields are

# required if using an application credential to authenticate. Some providers

# allow you to create an application credential to authenticate rather than a

# password.

[ application_credential_name: <string> ]

[ application_credential_id: <string> ]

# The application_credential_secret field is required if using an application

# credential to authenticate.

[ application_credential_secret: <secret> ]

# Whether the service discovery should list all instances for all projects.

# It is only relevant for the 'instance' role and usually requires admin permissions.

[ all_tenants: <boolean> | default: false ]
```

```
# Refresh interval to re-read the instance list.

[ refresh_interval: <duration> | default = 60s ]

# The port to scrape metrics from. If using the public IP address, this must

# instead be specified in the relabeling rule.

[ port: <int> | default = 80 ]

# TLS configuration.

tls_config:

  [ <tls_config> ]
```

Prometheus discovery service provides also file discovery approach where targets can be added on file and Prometheus will load automatically targets. This feature is crucial for automatizing Prometheus.

The next service allows retrieving targets from google instances. <gce_sd_config>.

Those are the available labels:

__*meta_gce_instance_id*: the numeric id of the instance

__*meta_gce_instance_name*: the name of the instance

__*meta_gce_label_<name>*: each GCE label of the instance

__*meta_gce_machine_type*: full or partial URL of the machine type of the instance

__*meta_gce_metadata_<name>*: each metadata item of the instance

__*meta_gce_network*: the network URL of the instance

__*meta_gce_private_ip*: the private IP address of the instance

__*meta_gce_project*: the GCP project in which the instance is running

__*meta_gce_public_ip*: the public IP address of the instance, if present

__*meta_gce_subnetwork*: the subnetwork URL of the instance

__*meta_gce_tags*: comma separated list of instance tags __*meta_gce_zone*: the GCE zone URL in which the instance is running

The configuration block is the follow:

```
# The information to access the GCE API.

# The GCP Project

project: <string>

# The zone of the scrape targets. If you need multiple zones use multiple

# gce_sd_configs.

zone: <string>

# Filter can be used optionally to filter the instance list by other criteria

# Syntax of this filter string is described here in the filter query parameter section:

# https://cloud.google.com/compute/docs/reference/latest/instances/list

[ filter: <string> ]

# Refresh interval to re-read the instance list

[ refresh_interval: <duration> | default = 60s ]

# The port to scrape metrics from. If using the public IP address, this must

# instead be specified in the relabeling rule.

[ port: <int> | default = 80 ]

# The tag separator is used to separate the tags on concatenation

[ tag_separator: <string> | default = , ]
```

The last discovery service that we will cover in this project is the one related to Kubernetes which is a container orchestrator <kubernetes_sd_configs>.

This service propose five groups of labels:

The first is labels related to Node (node can be seen as a vm)

__meta_kubernetes_node_name: The name of the node object.

__meta_kubernetes_node_label_<labelname>: Each label from the node object.

__meta_kubernetes_node_labelpresent_<labelname>: true for each label from the node object.

__meta_kubernetes_node_annotation_<annotationname>: Each annotation from the node object.

__meta_kubernetes_node_annotationpresent_<annotationname>: true for each annotation from the node object.

__meta_kubernetes_node_address_<address_type>: The first address for each node address type, if it exists.

The second set of labels are labels related to Pod (pods are Kubernestes' vocabulary for expressing one or many containers)

__meta_kubernetes_namespace: The namespace of the pod object.

__meta_kubernetes_pod_name: The name of the pod object.

__meta_kubernetes_pod_ip: The pod IP of the pod object.

__meta_kubernetes_pod_label_<labelname>: Each label from the pod object.

__meta_kubernetes_pod_labelpresent_<labelname>: truefor each label from the pod object.

__meta_kubernetes_pod_annotation_<annotationname>: Each annotation from the pod object.

__meta_kubernetes_pod_annotationpresent_<annotationname>: true for each annotation from the pod object.

__meta_kubernetes_pod_container_init: true if the container is an InitContainer __*meta_kubernetes_pod_container_name*: Name of the container the target address points to.

__*meta_kubernetes_pod_container_port_name*: Name of the container port.

__*meta_kubernetes_pod_container_port_number*: Number of the container port.

__*meta_kubernetes_pod_container_port_protocol*: Protocol of the container port.

__*meta_kubernetes_pod_ready*: Set to true or false for the pod's ready state.

__*meta_kubernetes_pod_phase*: Set to Pending, Running, Succeeded, Failed or Unknown in the lifecycle.

__*meta_kubernetes_pod_node_name*: The name of the node the pod is scheduled onto.

__*meta_kubernetes_pod_host_ip*: The current host IP of the pod object.

__*meta_kubernetes_pod_uid*: The UID of the pod object.

__*meta_kubernetes_pod_controller_kind*: Object kind of the pod controller.

__*meta_kubernetes_pod_controller_name*: Name of the pod controller.

The third group are the labels regarding Kubernetes services

__*meta_kubernetes_namespace*: The namespace of the service object.

__*meta_kubernetes_service_annotation_<annotationname>*: Each annotation from the service object.

__*meta_kubernetes_service_annotationpresent_<annotationname>*: "true" for each annotation of the service object.

__*meta_kubernetes_service_cluster_ip*: The cluster IP address of the service. (Does not apply to services of type ExternalName) __*meta_kubernetes_service_external_name*: The DNS name of the service. (Applies to services of type ExternalName)

__*meta_kubernetes_service_label_<labelname>*: Each label from the service object.

__*meta_kubernetes_service_labelpresent_<labelname>*: true for each label of the service object.

__*meta_kubernetes_service_name*: The name of the service object.

__*meta_kubernetes_service_port_name*: Name of the service port for the target.

__*meta_kubernetes_service_port_protocol*: Protocol of the service port for the target.

The next set of labels are labels related to endpoints

__*meta_kubernetes_endpoint_hostname*: Hostname of the endpoint.

__*meta_kubernetes_endpoint_node_name*: Name of the node hosting the endpoint.

__*meta_kubernetes_endpoint_ready*: Set to true or false for the endpoint's ready state.

__*meta_kubernetes_endpoint_port_name*: Name of the endpoint port.

__*meta_kubernetes_endpoint_port_protocol*: Protocol of the endpoint port.

__*meta_kubernetes_endpoint_address_target_kind*: Kind of the endpoint address target.

__*meta_kubernetes_endpoint_address_target_name*: Name of the endpoint address target.

The last group of labels are related to Kubernetes ingress __*meta_kubernetes_namespace*: The namespace of the ingress object.

__*meta_kubernetes_ingress_name*: The name of the ingress object.

__*meta_kubernetes_ingress_label_<labelname>*: Each label from the ingress object.

__*meta_kubernetes_ingress_labelpresent_<labelname>*: true for each label from the ingress object.

__*meta_kubernetes_ingress_annotation_<annotationname>*: Each annotation from the ingress object.

__*meta_kubernetes_ingress_annotationpresent_<annotationname>*: true for each annotation from the ingress object.

__*meta_kubernetes_ingress_scheme*: Protocol scheme of ingress, https if TLS config is set. Defaults to http.

__*meta_kubernetes_ingress_path*: Path from ingress spec. Defaults to /.

The below block is the configuration part regarding Kubernetes

# The information to access the Kubernetes API.

# The API server addresses. If left empty, Prometheus is assumed to run inside

# of the cluster and will discover API servers automatically and use the pod's

# CA certificate and bearer token file at /var/run/secrets/kubernetes.io/serviceaccount/.

[api_server: <host>]

# The Kubernetes role of entities that should be discovered.

role: <role>

# Optional authentication information used to authenticate to the API server.

```
# Note that `basic_auth`, `bearer_token` and `bearer_token_file` options are

# mutually exclusive.

# password and password_file are mutually exclusive.

# Optional HTTP basic authentication information.

basic_auth:

  [ username: <string> ]

  [ password: <secret> ]

  [ password_file: <string> ]

# Optional bearer token authentication information.

[ bearer_token: <secret> ]

# Optional bearer token file authentication information.

[ bearer_token_file: <filename> ]

# Optional proxy URL.

[ proxy_url: <string> ]

# TLS configuration.

tls_config:

  [ <tls_config> ]

# Optional namespace discovery. If omitted, all namespaces are used.

namespaces:

  names:

    [ - <string> ]
```

- Alert manager: This part handles alert sending after having defined rules. Alerts are sent after the violation of rules. Rules can be a simple expression of a composed expression (aggregation)

There are three ways of collecting metrics using Prometheus. The first and the most used is by embedding an exporter in the client application. The following schema describes the overall architecture of this model.

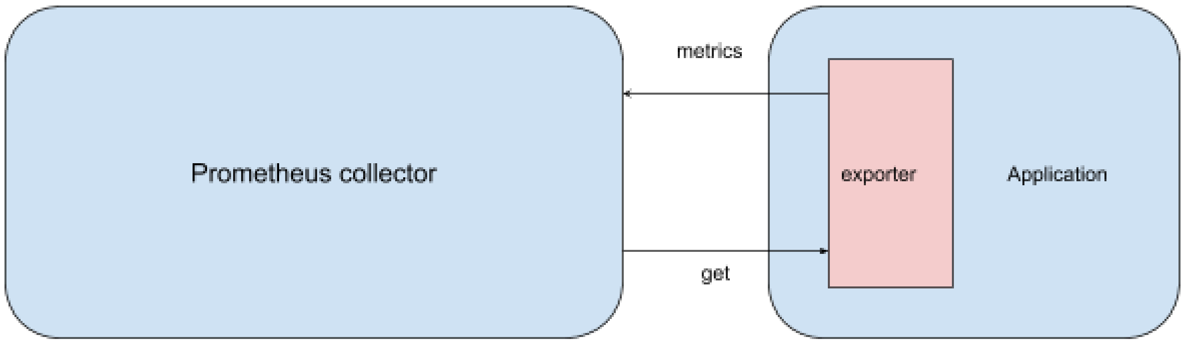


Fig. 1.2

The file `/etc/prometheus/prometheus.yml` should be properly configured in order to allow this functionality. The following shows basics parameters to be specified.

```
global:
  scrape_interval:     15s # By default, scrape targets every 15 seconds.

  # Attach these labels to any time series or alerts when communicating with
  # external systems (federation, remote storage, Alertmanager).
  external_labels:
    monitor: 'codelab-monitor'

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
  - job_name: 'prometheus'

    # Override the global default and scrape targets from this job every 5 seconds.
    scrape_interval: 5s

    static_configs:
      - targets: ['localhost:9090']
```

Fig. 1.3

One of the most important parameters of a metric collector is the interval of time metrics are gathered (*scrape_interval*). This interval is also called scraping time. This last is chosen according to the exporter and also the need of consumers in the system. If the scraping time is smaller than the interval of time metrics are requested for consumption, the collector will gather unnecessary data point [4]. If the scraping time is bigger than the interval of time metrics are requested, the decision units may miss some important data points.

job_name: The name of the data source
scrape_interval : This parameter determines the interval of time Prometheus collector queries the exporter for getting metrics.
targets: The endpoint (exporter) where metrics are exposed.

The second method is to collect measurements by federation of prometheus instances. There are different use cases for federation. Commonly, it is used to either achieve scalable Prometheus monitoring setups or to pull related metrics from one service's Prometheus into another.

Hierarchical federation

Hierarchical federation allows Prometheus to scale to environments with tens of data centers and millions of nodes. In this use case, the federation topology resembles a tree, with higher-level Prometheus servers collecting aggregated time series data from a larger number of subordinated servers.

For example, a setup might consist of many per-datacenter Prometheus servers that collect data in high detail (instance-level drill-down), and a set of global Prometheus servers which collect and store only aggregated data (job-level drill-down) from those local servers. This provides an aggregate global view and detailed local views.

Cross-service federation

In cross-service federation, a Prometheus server of one service is configured to scrape selected data from another service's Prometheus server to enable alerting and queries against both datasets within a single server.

For example, a cluster scheduler running multiple services might expose resource usage information (like memory and CPU usage) about service instances running on the cluster. On the other hand, a service running on that cluster will only expose application-specific service metrics. Often, these two sets of metrics are scraped by separate Prometheus servers. Using federation, the Prometheus server containing service-level metrics may pull in the cluster resource usage metrics about its specific service from the cluster Prometheus, so that both sets of metrics can be used within that server.

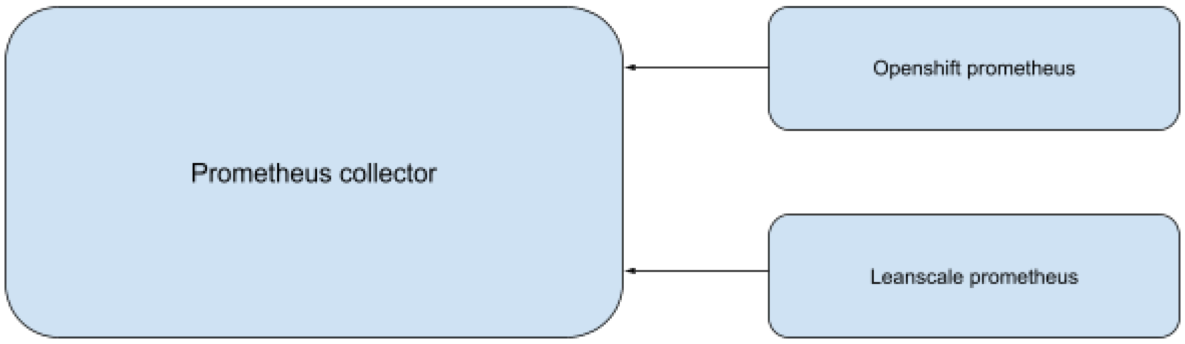


Fig. 1.4

Service discovery

Being able to target and monitoring an application requires, configuring at prometheus' side "job" information as described above. In case where an application can be deployed dynamically, modifying prometheus' config file is not efficient since it requires to restart prometheus. To solve this limitation, prometheus provide a dynamic discovery mechanism where prometheus verify periodically a discovery file in order to capture modifications and apply them dynamically.

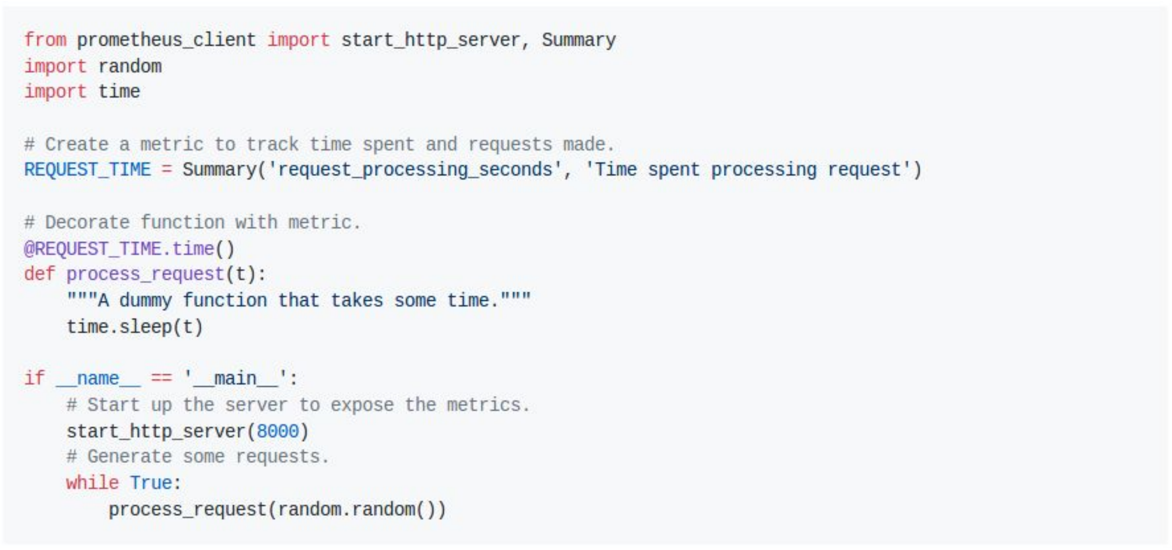
```
scrape_configs:
- job_name: 'node'
  file_sd_configs:
  - files:
    - 'targets.json'
```

Service discovery capability is used by added these above lines in the prometheus.yml file
And the content of "targets.json" is the follow:

```
[
  {
    "labels": {
      "job": "node"
    },
    "targets": [
      "localhost:9100"
    ]
  }
]
```

Exporter

An exporter is a software component inserted in an application with the purpose of collecting data (metrics) then exposing them. Prometheus uses http protocol for requesting metrics. Thus , the exporter should implement an endpoint where metrics and others information related will be accessed. Prometheus offers libraries for implementation of exporters in different programming languages. Natively prometheus libraries provides an http server which can be included in an application. In case the application already have an http server, the collection of metrics can be customized using the existing http server.



```python
from prometheus_client import start_http_server, Summary
import random
import time

# Create a metric to track time spent and requests made.
REQUEST_TIME = Summary('request_processing_seconds', 'Time spent processing request')

# Decorate function with metric.
@REQUEST_TIME.time()
def process_request(t):
    """A dummy function that takes some time."""
    time.sleep(t)

if __name__ == '__main__':
    # Start up the server to expose the metrics.
    start_http_server(8000)
    # Generate some requests.
    while True:
        process_request(random.random())
```

The above snapshot describes a very basic prometheus exporter. In this example, the exporter is exposing the time spent processing request. The type of this metric is summary which means that all observations (data points) collected by prometheus will be aggregated. The following aggregation will take place:
Count: the number of observations
Sum: the sum of all observations.
The exporting is using the port 8000 through the native http server provided by prometheus libraries. Metrics will be available on *http://hostname:8000/metrics*

We have to mention that Prometheus has four metric types:

1. *Counter*: A counter is a cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart. For example, you can use a counter to represent the number of requests served, tasks completed, or errors.

2. *Gauge:* A gauge is a metric that represents a single numerical value that can arbitrarily go up and down.Gauges are typically used for measured

values like temperatures or current memory usage, but also "counts" that can go up and down, like the number of concurrent requests

3. *Summary:* Similar to a histogram, a summary samples observations (usually things like request durations and response sizes). While it also provides a total count of observations and a sum of all observed values, it calculates configurable quantiles over a sliding time window*.*

4. *Histogram:* A histogram samples observations (usually things like request durations or response sizes) and counts them in configurable buckets. It also provides a sum of all observed values.

Prometheus’ SDK offers capability of integration its classes to a different http server. This is important for exporting metrics in applications such as apache server, nginx based service.

Exporter is the best approach for collecting metrics, however, this technique cannot satisfied every use case. We will describe two cases where the exporter approach cannot be applied:

- Prometheus uses http server for exposing metrics. Since an http server requires a loop for listening connexion, components implementing a blocking connexion others than http protocol oriented, cannot manage a second listener. We will mention a queueing component consumer.

- Prometheus requires a static settings in order the scrape metrics from exporter register. This implies a static hostname or IP address, this is not achievable all the time for generated component.

I.5 Storing

We are using prometheus in this project and this last has limited retention time which is not enough for a big data environment. We need to provide a mechanism by which time series will be stored for historical purpose and ad-hoc access. We have to emphasize that the platform can host throusant of applications where each produces many metrics per second. Thus the environment will have to handle more 1000 writing per second. Another aspect to take in account for an efficient choice and storing management system is the distributed nature of application in the platform. The platform can handle many nodes where nodes are not necessarily located in the same physical environment. This introduces the distributed character of the platform, thus the distributed nature that must support the storing system.
The storing should be compatible with the visualization system in order to be used easily as metrics source for metrics visualization.

Thus the need of using a different time series storage, we will be using Elasticsearch[5] for its rich features such as : query language, distribution behavior, high throughput.
As we said in the abstract, we aim to optimize the storing to avoid storing unnecessary data points.
The monitoring must also provide a prediction mechanism. The most efficient prediction mechanisms use machine learning techniques which require a considerable set of data in order to build a good model. We are faced with a tradeoff where from one side we want to avoid saving unnecessary data by on the other hand, these entries could contain some pattern relevant for the machine learning model.

I.6 Exposition

Metrics could be exposed using an ad-hoc mode and a streaming mode. We will cover both approaches in this paper. The ad-hoc mode is mostly used for historical purpose whilst the streaming mode is used for delivering metrics in real-time. The most suitable way of implementing this functionality is by using a queueing system.

I.7 Queuing System

In order to allow a real-time consumption, a queuing mechanism is the most suitable solution. A queuing system or massaging system provides an asynchronous communications protocol, meaning that the sender (producer) and receiver (consumer) of the message do not need to interact with the message queue at the same time. Messages placed onto the queue are stored until the recipient retrieves them. Message queues have implicit or explicit limits on the size

of data that may be transmitted in a single message and the number of messages that may remain outstanding on the queue.
In a cloud environment where a system composed of many components, components integration (communication between components) is a hard task since every component may have a different architecture and specifications. Taking into account the amount of data, a queuing system seems to be a very approach since messages (requests) stays in the queue waiting to be consumed by the handler component.This feature prevents timeout that could happen in http based communication.
In this project we will focus on the most popular nowadays: RabbitMQ and Apache Kafka. Each has its own origin story, design intent, uses cases where it suits, integration capabilities and developer experience. Origins are revealing about the overall design intent for any piece of software, and make a good entry point. However it’s important to note that in this project, our aim is to compare the two around the monitoring engine on a cloud environment.
RabbitMQ is a “traditional” message broker that implements a variety of messaging protocols. It was one of the first open source message brokers to achieve a reasonable level of features, client libraries, dev tools, and quality documentation.

RabbitMQ was originally developed to implement AMQP, an open wire protocol for messaging with powerful routing features. While Java has messaging standards like JMS, it’s not helpful for non-Java applications that need distributed messaging which is severely limiting to any integration scenario, microservice or monolithic. With the advent of AMQP, cross-language flexibility became real for open source message brokers.

Apache Kafka is developed in Scala and started out at LinkedIn as a way to connect different internal systems. At the time, LinkedIn was moving to a more distributed architecture and needed to reimagine capabilities like data integration and real time stream processing, breaking away from previously monolithic approaches to these problems. Kafka is well adopted today within the Apache Software Foundation ecosystem of products and is particularly useful in event-driven architecture.

RabbitMQ is designed as a general purpose message broker, employing several variations of point to point, request/reply and pub-sub communication styles patterns. It uses a smart broker / dumb consumer model, focused on consistent delivery of messages to consumers that consume at a roughly similar pace as the broker keeps track of consumer state. It is mature, performs well when configured correctly, is well supported (client libraries Java, .NET, node.js, Ruby, PHP and many more languages) and has dozens of plugins available that extend it to more use cases and integration scenarios.

Apache Kafka includes the broker itself, which is actually the best known and the most popular part of it, and has been designed and prominently marketed towards stream processing scenarios. In addition to that, Apache Kafka has recently added Kafka Streams which positions itself as an alternative to streaming platforms such as Apache Spark, Apache Flink, Apache Beam/Google Cloud DataFlow and

Spring Cloud Data Flow. The documentation does a good job of discussing popular use cases like Website Activity Tracking, Metrics, Log Aggregation, Stream Processing, Event Sourcing and Commit logs. One of those use cases it describes is messaging, which can generate some confusion. So let's unpack that a bit and get some clarity on which messaging scenarios are best for Kafka for, like:

- Stream from A to B without complex routing, with maximal throughput (100k/sec+), delivered in partitioned order at least once.
- When your application needs access to stream history, delivered in partitioned order at least once. Kafka is a durable message store and clients can get a "replay" of the event stream on demand, as opposed to more traditional message brokers where once a message has been delivered, it is removed from the queue.
- Stream Processing
- Event Sourcing

RabbitMQ is a general purpose messaging solution, often used to allow web servers to respond to requests quickly instead of being forced to perform resource-heavy procedures while the user waits for the result. It's also good for distributing a message to multiple recipients for consumption or for balancing loads between workers under high load (20k+/sec). When your requirements extend beyond throughput, RabbitMQ has a lot to offer: features for reliable delivery, routing, federation, HA, security, management tools and other features. Let's examine some scenarios best for RabbitMQ, like:

- Your application needs to work with any combination of existing protocols like AMQP 0-9-1, STOMP, MQTT, AMQP 1.0.
- You need a finer-grained consistency control/guarantees on a per-message basis (dead letter queues, etc.) However, Kafka has recently added better support for transactions.
- Your application needs variety in point to point, request / reply, and publish/subscribe messaging
- Complex routing to consumers, integrate multiple services/apps with non-trivial routing logic

RabbitMQ can also effectively address several of Kafka's strong uses cases above, but with the help of additional software. RabbitMQ is often used with Apache Cassandra when application needs access to stream history, or with the LevelDB plugin for applications that need an "infinite" queue, but neither feature ships with RabbitMQ itself.

1.8 Prediction

The prediction is a feature of a monitoring engine which allows to forecast from some data the future value and enabling a proactive behavior of the platform. Information collected in the platform has the characteristic of a time series which is a series of data points indexed (or listed or graphed) in time order. Most commonly, a time series is a sequence taken at successive equally spaced points in time[X].

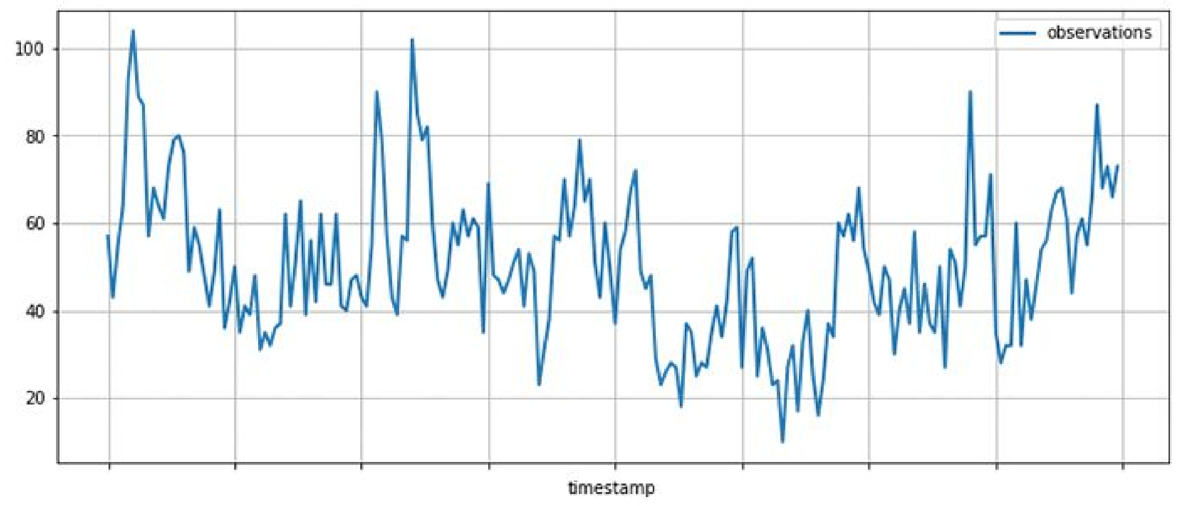


Fig.

Time series prediction or time series forecasting is a hot topic which has many possible applications, such as stock prices forecasting, weather forecasting, business planning, resources allocation and many others. Even though forecasting can be considered as a subset of supervised regression problems, some specific tools are necessary due to the temporal nature of observations. In our context, time series prediction will be used to perform the main operations crucial for a monitoring engine. First is to provide proactive behavior of the platform, second is to enable the detection of relevant features and enable detection of bottleneck.

2. State of the art

The performance of a Cloud service has already been addressed in bibliography. For example, de Vaulx et al. developed a model for the performance of the Cloud at an application level (Quality of Service, availability, reliability, etc.). This is consistent with the services offered by most Cloud providers, which ensure the user a minimum availability time during the lease. On the other hand, the cloud provider is interested in optimizing the performance and utilization of the data centre at a system level, (manageability, fault tolerance, energy consumption, etc.).
These two levels of evaluation are opposed (the user pushes for a better QoS, while the provider requires a more efficient use of resources), and there is not a standard approach to unifying the concerns of both. A compromise between the parts is usually the approach, economical (the provider makes a worse use of resources and the price to the user is increased), moral [6], etc. However, to the best of our knowledge, there is not a centralized approach to ensure the performance of metrics at both levels (application and system level) simultaneously.

To be able to maintain a good quality and perform best adaptation based on the change that could happen in a system, metrics need to be taken contently and expose to the component involved in the evaluation of quality and adaptation. In the context of big datastack, tracking information will be performed by the Triple monitoring engine. Three different groups of metrics need to track: infrastructure information, data operation (data produced by applications running on the platform) and all data involved in database transactions.
Since these metrics are produced by applications with different purposes, specifications, functionalities and technologies, two approaches will be used, the first is to use probe to directly ingest metrics into the monitoring collector. The second approach is to provide a sanitizer to prepare metrics conforming with the specification of the collector and ingest them. This sanitizer will act as a unified API. The triple monitoring engine has an input REST API which is an entry point of the system and an output REST API for exposing data to all applications data consumer. The monitoring should provide an efficient and fast way of transferring metrics from the input to the manager that handle all the logic of the engine. The big number of metrics from different sources must be organized chronologically and presented to a correct format for their visualization. We've been interested by two main technologies:
Prometheus is a technology for monitoring management, which includes metrics collection
facilities. This technology will be very convenient for the following reasons 19 :

- Powerful queries: A flexible query language (NoSQL based) allows slicing and dicing of collected time series data.
- Efficient storage: Prometheus stores times series in memory and on local in an efficient custom format. Scaling is achieved by function shading and federation.

- Extensive integration: Many existing exporters allow bringing data from third-party application to its collector.
- Push gateway: In case it’s impossible to scrape metrics (using probe), metrics can be exposed to the Prometheus collector by this mechanism.

The manager needs a persistent connection with the output REST API, a connection oriented
based technology will be used. RabbitMQ will be very convenient because of the following :

- Availability in many languages and platforms.
- Asynchronous Messaging: Supports multiple messaging protocols, message queuing, delivery acknowledgement, flexible routing to queues, multiple exchange type. Those features allow to easily a publish/subscribe mechanism, high-speed asynchronous I/O engines, in a tiny library.
- Distributed Deployment: Deploy as clusters for high availability and throughput; federate across multiple availability zones and regions.

Persistent data need to be stored for later use, since all REST API within triple monitoring engine use JSON format and metrics don't have the same structure because of their respective origin, a convenient technology for saving these data will be using a database that handle JSON format to facilitate data transfer within the triple monitoring engine and to allow polymorphism. Based on the amount of data arriving per second and the huge quantity of operation that need to be perform MongoDB will be very efficient.
As said before, the triple monitoring engine provides two REST interfaces.

- The first has the goal of receiving data from different sources and sending them to the Netdata collector (plugin). This interface will be the input of monitoring engine. The API keeps data in memory until they are consumed by the plugin. Applications (data producers) will have access to this API for sending their measurements.
- The second interface provides the output of the monitoring engine to applications (consumers). This interface has two kinds of connection to serve results: a REST API and a Publish/Subscribe mechanism that is connection-oriented service.

Netdata is a system for health and performance monitoring of distributed real-time systems. It provides real-time insights of everything happening on the system it runs (including applications such as web and database servers), using interactive web dashboards [7]. Netdata main capabilities are gathering data from different sources and exposing them through a REST API. Netdata architecture is extensible through plugins to read measurements (metrics) from different sources. In Figure 3 , the component named “BigDataStack plugin” is an adapter that needs to be deployed to ingest data into Netdata. Since each application/source has its own specificities based on its functionalities, metrics could be different. Each application/source will expose its metrics to the monitoring collector API.

3. Monitoring Approach

3.1 Introduction

The monitoring engine manages and correlates/aggregates monitoring data from different levels to provide a better analysis of the environment, the application and data; allowing the orchestrator to take informed decisions in the adaptation engine. The engine collects data from three different sources:

- Infrastructure resources of the compute clusters such as resource utilisation (CPU and RAM), availability of the hosts, data sources generation rates and windows. This information allows the taking of decisions at a low level. These metrics are directly provided by the infrastructure owner or through specific probes, which track the quality of the available infrastructures. We are using federation of prometheus's instances in order to ingest those metrics into the triple monitoring engine. In order to do not save unnecessary metrics, we will be using Prometheus filtering feature which allows to select jobs (metrics sources) related to our needs.

- Application components such as application metrics, data flows across application components, availability of the applications etc. This information is related directly to the data-driven services, which are deployed in the infrastructure. These metrics are associated with each application, and they should be provided by those applications.

- Data functions/operations such as data analytics, query progress tracking, storage distribution, etc. This is a mix of data and storage infrastructure information providing additional information for the "data-oriented" infrastructure resources.

The component will cover both raw metrics (direct measurements provided by the infrastructure deployed sensors or external measurement systems like the status of infrastructure) and aggregated metrics (formulas to exploit metrics already collected and produce the respective aggregated measurements that can be more easily used for QoS tracking). The collection of metrics will be based on both solutions: the direct probes (exporters) in the system that should be monitored and the direct collection of the data from the monitoring engine.

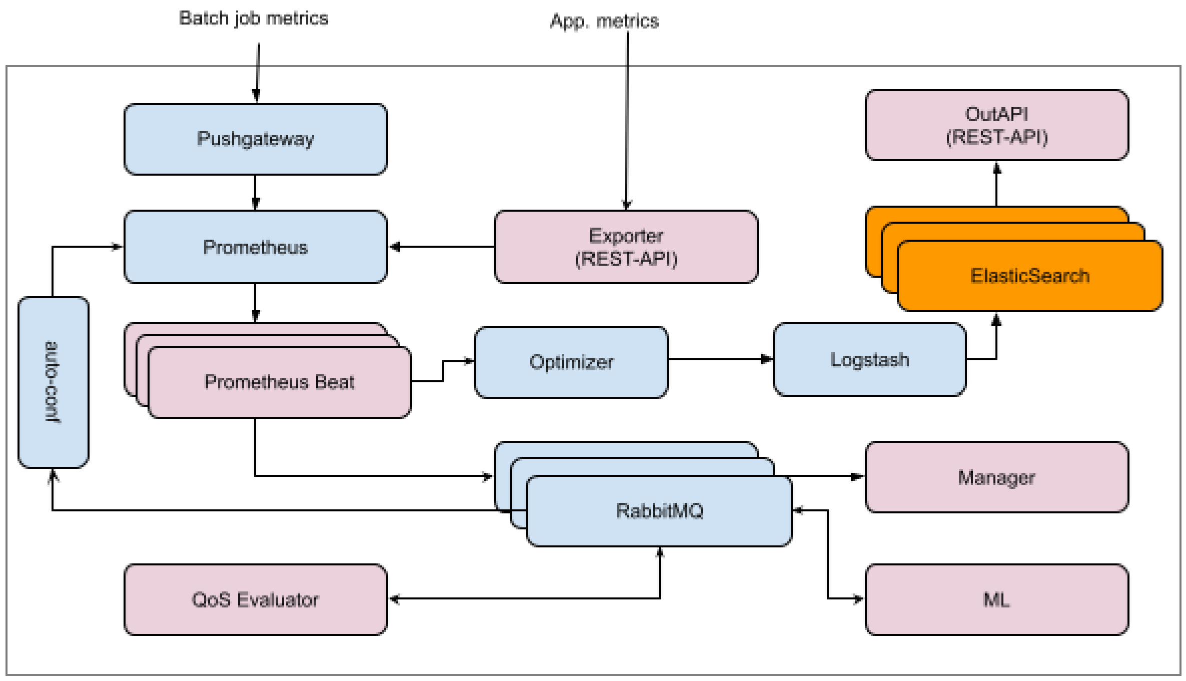
- The probe approach will cover the information systems, where the platform will be able to deploy and collect direct information. In this case, the orchestration engine must manage the deployment of the necessary probes. This approach can cover other cases, where the probe is included directly in the application, and the orchestration only needs to deploy the associated application, which can provide the metric information to the monitoring engine.

- The direct collection will cover the scenarios where the platform cannot deploy any probe, but the infrastructures or the applications expose some information regarding these metrics. In this case, the monitoring engine will be responsible for collecting the metrics data that are exposed by a third party.

The database is responsible for persisting all the data. The database will be dimensioned depending on historical requirements, the kind of aggregation and the expected volume of data produced by the metrics. The monitoring DB component will either be a separate component or provide all the information to the global decision tracker of the architecture.

After collecting and processing the data, the monitoring engine will be responsible for notifying other components when an event happens based on the metrics that it is tracking and specific attributes such as computing, network, storage or application level. Moreover, it will expose an interface to manage and query the content. We will be covering also proactive violation giving the decision component future event for better management of platform resources. The proactive functionality is not fully implemented in this project, however, the monitoring has the capabilities of building a dataset by detecting different relevant metrics related to an SLO. The monitoring engine will also cover dynamic prometheus target discovery allowing the auto configuration of an application. In order to optimize the storage system, we will be defining the utility of each metric, thus finding the suitable interval of time this metric can be stored.

3.2 General architecture

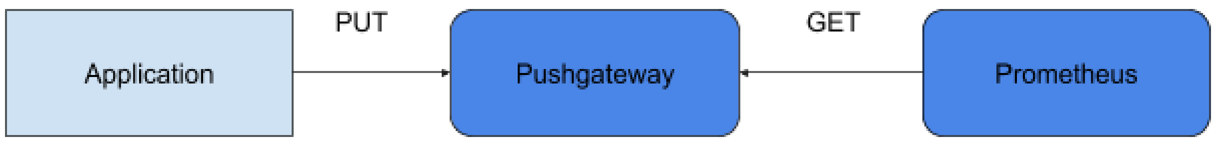


"Figure 2.1"

The Tripe Monitoring Engine will be based on the Prometheus monitoring solution (see [11] for more details) and is composed of the following components:

3.2.1 Pushgateway

This component collects metrics from batch job or ephemeral application then ingest then into prometheus. Any component of the platform can submit its metrics using the PUT method of the http protocol. The pushgateway listens to the port 9091. This component is special for jobs such spark, hadoop. The pushgateway accepts metrics over http by PUT method, store them in memory then expose them to Prometheus.



Here is an example demonstrating the use of this component for collecting Spark metrics.

In order to implement this scenario, we use sparkMeasure[X] which is a simplified API for collecting spark measure and add prometheus client functionality for having the possibility to send metrics to Prometheus. After each spark job's execution a set of metrics are collected by sparkMeasure API from the execution layer (JMX).

The above list is the metrics collected by sparkMeasure.

numStages

sum(numTasks)

elapsedTime

sum(stageDuration)

sum(executorRunTime)

sum(executorCpuTime)

sum(executorDeserializeTime)

sum(executorDeserializeCpuTime)

sum(resultSerializationTime)

sum(jvmGCTime)

sum(shuffleFetchWaitTime)

sum(shuffleWriteTime)

max(resultSize)

sum(numUpdatedBlockStatuses)

sum(diskBytesSpilled)

sum(memoryBytesSpilled)

max(peakExecutionMemory)

sum(recordsRead)

sum(bytesRead)

sum(recordsWritten)

sum(bytesWritten)

sum(shuffleTotalBytesRead)

sum(shuffleTotalBlocksFetched)

sum(shuffleLocalBlocksFetched)

sum(shuffleRemoteBlocksFetched)

sum(shuffleBytesWritten)

sum(shuffleRecordsWritten)

These metrics are aggregation since the spark execution layer can create more than one worker for a single job.

They need to be parsed and exported in the following form:

num_stages

num_tasks

elapsed_time

stage_duration

executor_run_time

executor_cpu_time

executor_deserialize_time

executor_deserialize_cpu_time

result_serialization_time

jvm_gc_time

shuffle_fetch_wait_time

shuffle_write_time

result_size

num_updated_block_statuses

disk_bytes_spilled

memory_bytes_spilled

peak_execution_memory

records_read

bytes_read

records_written

bytes_written

shuffle_total_bytes_read

shuffle_total_blocks_fetched

shuffle_local_blocks_fetched

shuffle_remote_blocks_fetched

shuffle_bytes_written

shuffle_records_written

The particularity of the pushgateway that needs to be taken into consideration before using it is the fact the pushgateway keeps the last set of metrics exported. This implies that after the execution of the first job. The Pushgateway will always expose the last result to Prometheus until it receives a new set of results.

Here is the code sample if this example: *main.py*

```
from sparkmeasure import *
from pyspark import SparkContext
from pyspark.sql import SQLContext
from pyspark.sql import SparkSession
import random, re, time
from prometheus_client import CollectorRegistry, Gauge, push_to_gateway

class Spark():
        def __init__(self):
        self.spark_session = SparkSession.builder.appName('sql executor').getOrCreate()
        self.df = None
        self.path = "file:/home/jean-didier/Projects/bigdatastack/spark/sample.csv"

        def loadDF(self):
        self.df = self.spark_session.read.format("csv").option("header", "true").load(self.path)
        self.df.registerTempTable("house")
        self.spark_session.sql("select * from house").show()

        def execute(self,sql):
        result = self.spark_session.sql(sql)
        result.show()
        def getSession(self):
        return self.spark_session

class Runner():
        def __init__(self):
        self.interval = 10
        self.sqls = ["select * from house","select * from house where city='SACRAMENTO'","select
count(*) as nmb from house where state='CA'"]
        self.stop_running = False
        self.prefix = "spark_sql_"
```

```
def getNameAndValue(self,line):
# for str(metric) , return _prefix_str_metric
# for metric, return _prefix_metric
index = line.index("=>")
line_title = line[:index]
metric_name = ""
try:
start = line_title.index("(")
end = line_title.index(")")
metric_name = self.prefix+line_title[:start]+"_"+line_title[start+1:end]
except:
metric_name = self.prefix+ re.findall(r'[a-zA-Z]+',line_title[:index])[0]
value_metric = int(re.findall(r'[0-9]+',line)[0])
return (metric_name,value_metric)

def stop(self):
self.stop_running = True

def start(self):
spark = Spark()
spark.loadDF()
time.sleep(2)
print("Session started")
while not self.stop_running:
stage = StageMetrics(spark.getSession())
stage.begin()
spark.execute(self.sqls[random.randint(0,len(self.sqls)-1)])
stage.end()
##############################################################
registry = CollectorRegistry()
list_report = stage.readreport().split("\n")
for report in list_report:
if "=>" in report:
        name, value = self.getNameAndValue(report)
        g = Gauge(name,name, ['engine'], registry=registry)
        g.labels('sql_executor').set(value)
push_to_gateway('localhost:9091', job='spark_sql', registry=registry)
time.sleep(self.interval)
```

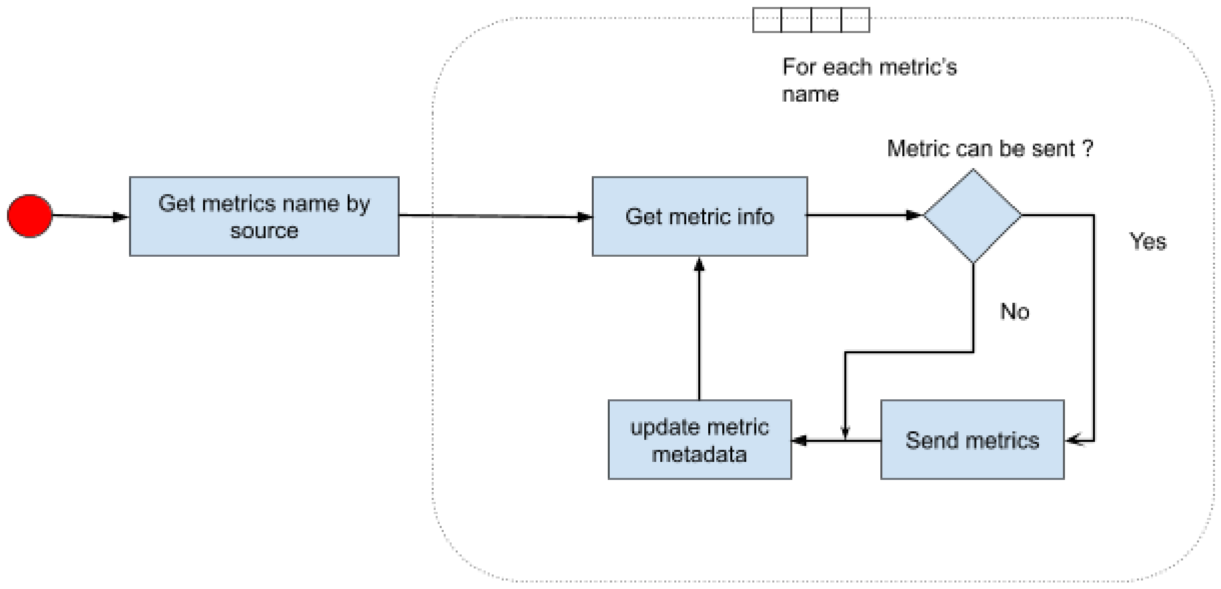
```
if __name__=="__main__":
        runner = Runner()
```

The execution of the script requires the java package *spark-measure.assembly.jar* which can be downloaded from the repository of the spark measure project or built from the same repository.

3.2.2 Prometheus Beat

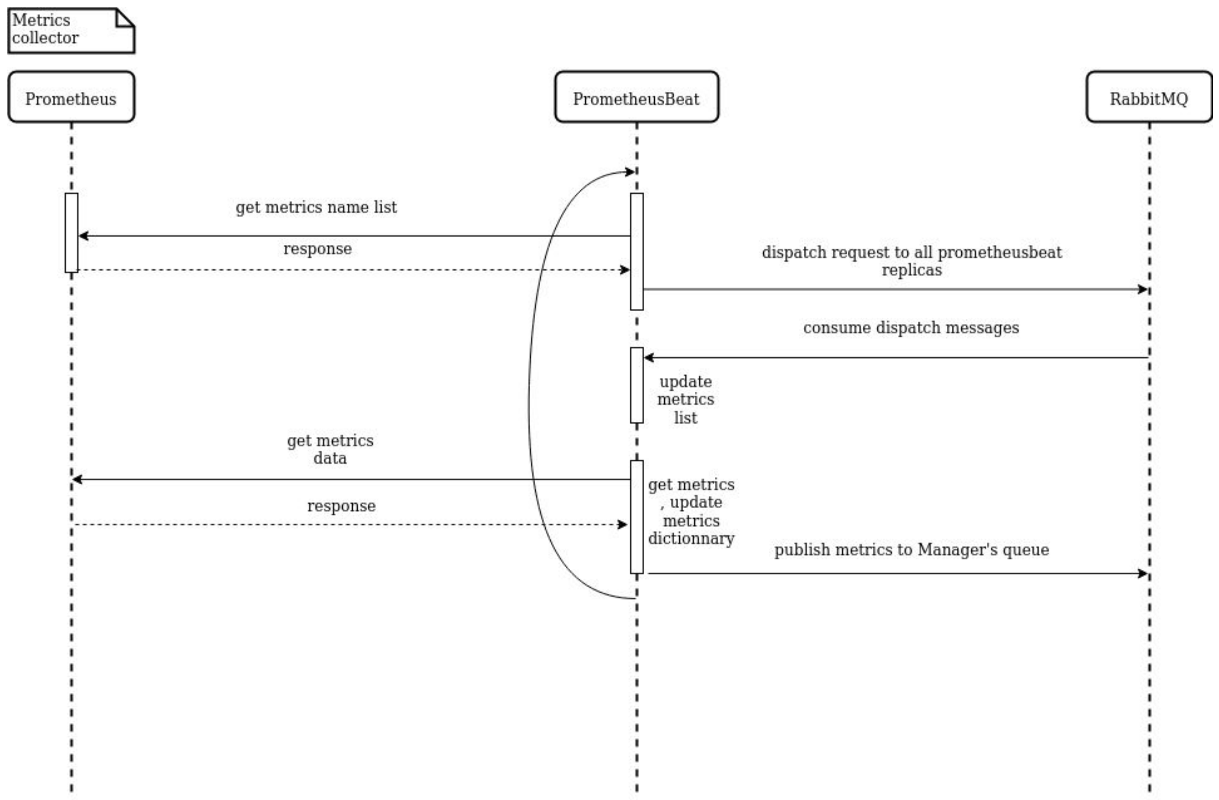
Since the retention period of prometheus is limited [1]. The use of an external storage is crucial for storing large amounts of time series data. Prometheus Beat reads periodically metrics from Prometheus then send to logstash and RabbitMQ. The period (sleeping time) can be configured based on the use case. However, this sleeping time is the main factor of the delay of the monitoring engine. The default value is 3 seconds. Prometheus Beat has the capability to send metrics to an endpoint and a queuing system. Presently this component is supporting RabbitMQ brokers. In order to not collect unnecessary metrics, Prometheus Beat offers the functionality of selecting prometheus' sources from which metrics will be read. Knowing that new metric can be added on running time, Prometheus beat updates its list containing metric's name each 10 minutes. This time can be modified by setting up the preferred value at the corresponding environment variable.

Activity diagram



"Fig.2.2"

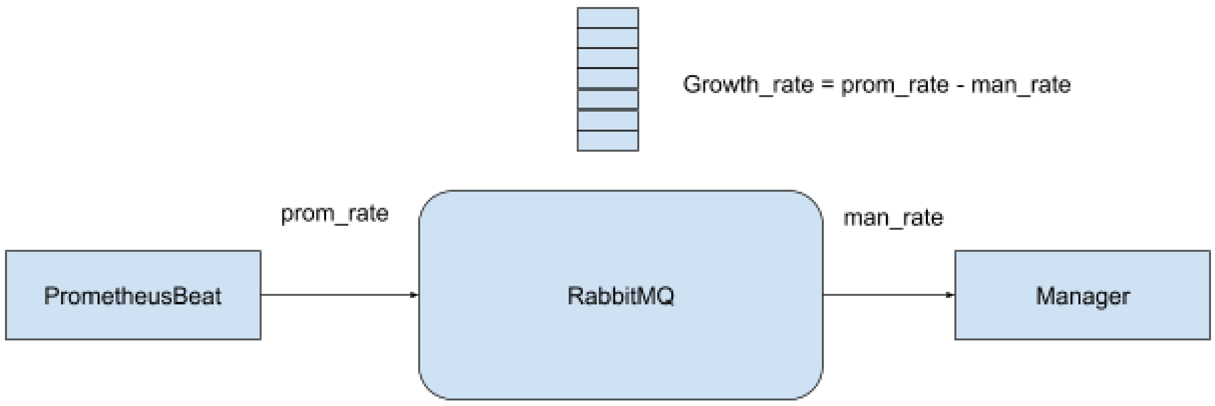
Sequence diagram


Metrics collector
Prometheus
PrometheusBeat
RabbitMQ
get metrics name list
response
dispatch request to all prometheusbeat replicas
consume dispatch messages
update metrics list
get metrics data
response
get metrics , update metrics dictionnary
publish metrics to Manager's queue

“Fig.2.3”

Building this component requires paying attention to two main situations. First, metrics are data points which are added to the collector (Prometheus) with time. Retrieving metrics from prometheus returns all metrics collected since the starting time. We need to provide mechanisms in order to not ingest into logstash duplicated data points. The approach used is to keep metric’s metadata (time, labels) in a dictionary and ingesting only metric which have the time higher than the previous kept. This feature requires the identification of each metric knowing many metrics from different sources could have the same name. For differentiating metrics we need to consider its label so we are building a hash composed by the name of the metric and all labels related to that metric. We said previously the monitoring delay depends mainly on the sleeping of prometheus beat. We have to emphasize that the request time (to prometheus) and the delivery time (sending request to logstash and RabbitMQ) impacts the delay that we want to minimize. The request time and the delivery time depends on the number of metrics collected. Otherwise these times grow considerably with the amount of metrics. The approach used is to build this component in a distributed manner such that all replicas could share tasks and then reduce the delay created by this component. There is no point of failure in the model we used. Replicas exchange messages in order to vote master when needed.

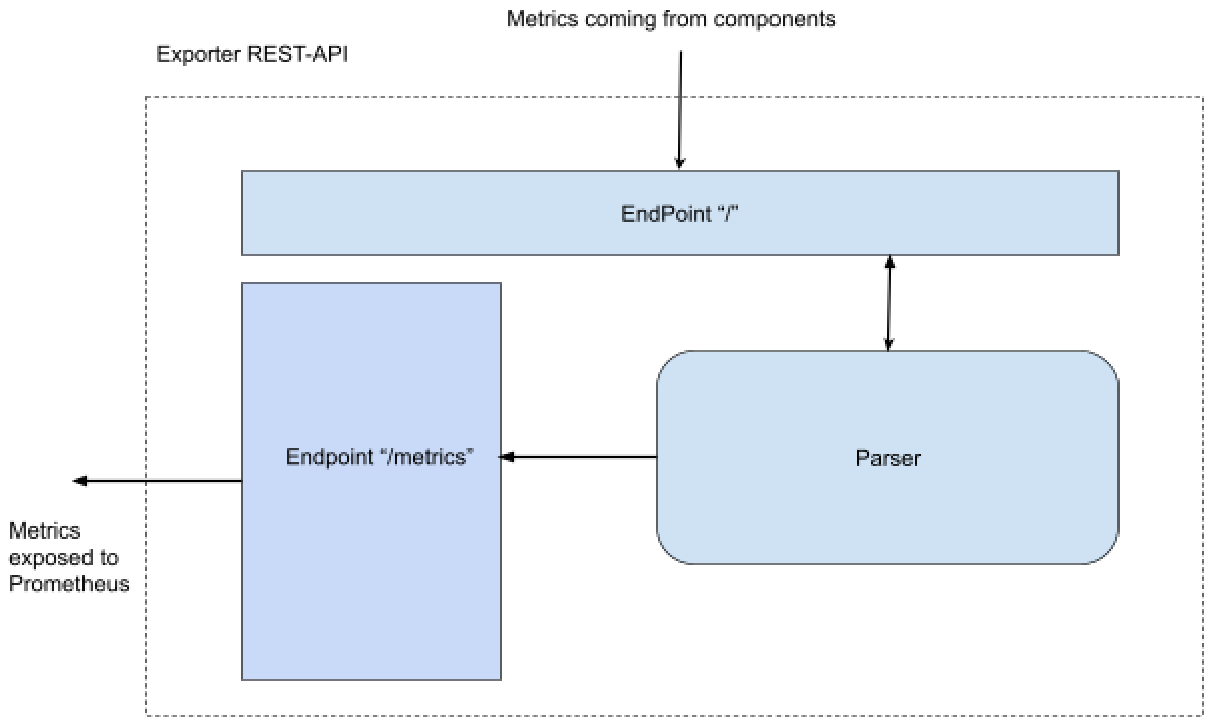
Using a queuing system is very efficient when dealing with big data, however one challenge has to be solved in order to ensure the good functioning of the system. This challenge is shown in the figure below.



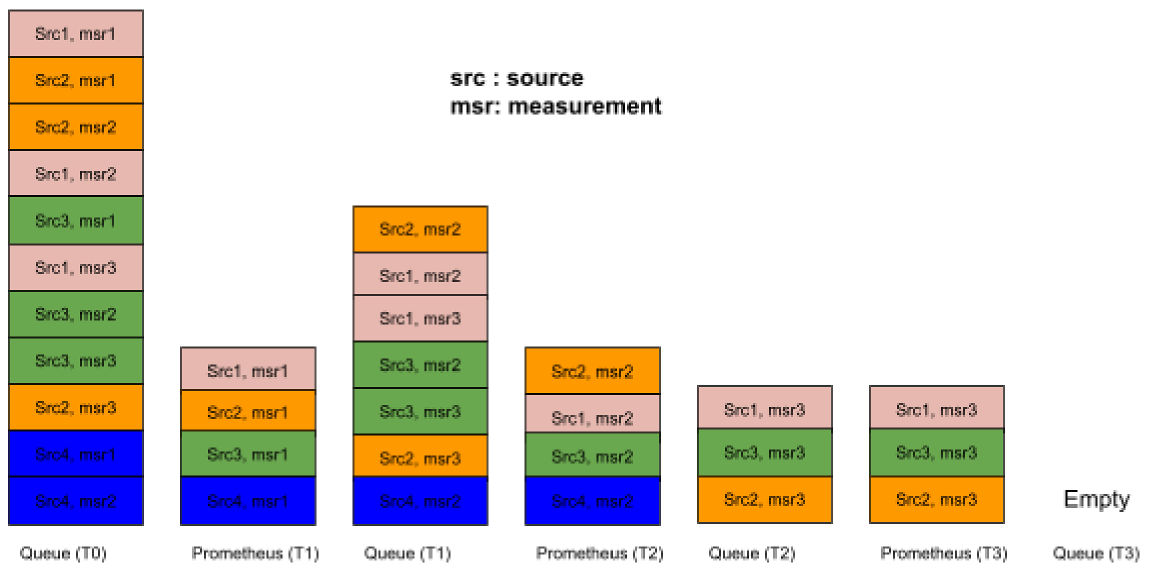
The ideal scenario will be to keep the growth_rate near to zero in order to not retain a huge amount of messages in the queue (not recommended). We have to build a mechanism by which we will be able to control the flow between the publisher (PrometheusBeat) and the consumer (Manager). This functionality has been implemented by collecting RabbitMQ metrics regarding the number of messages in the queue. We have set a threshold (max number of messages to retain in the queue). When this number reached, PrometheusBeat stopped publishing messages onto the manager's queue for some amount of time. This amount of time has been chosen by estimating the minimum consuming rate of the manager.

3.2.3 Exporter REST-API

The monitoring provides a component in order of collecting metrics where the use of the exporter is not possible. This component (Exporter - REST API) is a rest api which is listening on the port 55671 and exposes metrics received to Prometheus. There is another scenario where the use of this component is crucial for a monitoring engine. This is related to Prometheus's collection model. Prometheus needs information about the application's endpoint (hostname or ip and port) in order to collect metrics from that application. This requires the platform to assign a specific hostname or ip to an application. On a data-driven platform where application can be scaled horizontally, assigning an hostname or fixed IP to an application can be very expensive. Therefore, The REST exporter allows to reduce the amount of the IP or hostname that an environment can allocate.



The REST waits for metrics in json format parse them , then expose them to Prometheus. The json must contain labels in order to distinguish metrics origines.



The above picture describes the logic by which relies on the REST exporter. Since the component is receiving metrics from different sources. It's important to organize in fear way , those measurements will be exposed to Prometheus. Each source in the list needs to be exposed at the collection phase. For that, we have to provide a method for differentiating sources. Since two metrics coming from different sources can have the same name, the best strategy for distinguishing those measurements is by taking into account all labels related to that data point. Therefore, we are building a hash where the input is the all labels accompanying the data point.

API Specification

Request: send metrics

Method: *POST*

Endpoint: /

Data: *{“metrics”:{“metrics_name_1”:”value1”,”metrics_name_2”:”value2”},”labels”:{ “label”:”value”}}*

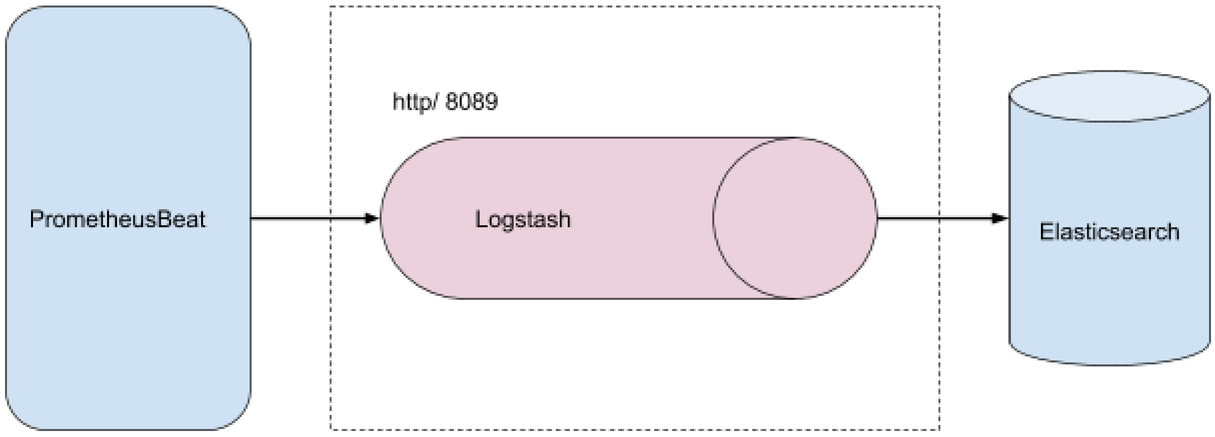
Response:

Success: *code(200), message(success)*

Error: *code(500), message(error parameters)*

3.2.4 Logstash

Logstash is an opensource tool, data processing pipelines for managing events and logs. Logstash can receive data from different sources.



This component is used in the monitoring engine for its capability of handling very high velocity information. Metrics read by Prometheus beat are ingested into logstash over http. Beats ingested have the following format:

{"name": "memory", "beat": {"version": "1.0", "name": "prometheusbeat", "hostname": "prometheusbeat"}, "@timestamp": "2019-10-09T10:00:38.457Z", "labels": {"engine": "triple_monitoring_engine", "instance": "exporter:55671", "job": "exporter", "component": "exporter"}, "value": "6656000", "time": 1570615238.457, "@version": "1"}

Logstash adds http header element in the JSON document for inserting into elasticsearch for storage. The output of logstash has the following format:

{"name": "memory", "beat": {"version": "1.0", "name": "prometheusbeat", "hostname": "prometheusbeat"}, "@timestamp": "2019-10-09T10:00:38.457Z", "labels": {"engine": "triple_monitoring_engine", "instance": "exporter:55671", "job": "exporter", "component": "exporter"}, "value": "6656000", "headers": {"http_accept": "/*", "content_length": "430", "http_user_agent": "python-requests/2.21.0", "http_version": "HTTP/1.1", "connection": "keep-alive", "request_method": "POST", "http_host": "logstash:8089", "content_type": "application/json", "x_requested_with": "Python requests", "accept_encoding": "gzip, deflate", "request_path": "/"}, "time": 1570615238.457, "@version": "1"}*

Logstash requires a configuration in order to initialized pipeline. The configuration used for our purpose is the following:

```
input {

  http {

    host => "${LOGSTASHHOST}"

    port => 8089

  }

}

filter{

  mutate {

    remove field => [ "host" ]

  }

}

output{

  elasticsearch {

    hosts => ["${ELASTICSEARCHHOST}:9200"]

    index => prometheus

  }

}
```

Logstash’s configuration file has three main parts. The first is the input, where data are coming from. The second is a filter, this is particularly useful when we want to process data before sending them in the output. The last part is the output which is elasticsearch for our case.

3.2.5 Monitoring Interface

This is responsible for exposing the interface to allow other components to communicate. The interface will manage two ways of interaction with other components: i) exposing a REST API that will enable other components to know specific information, for example, if other component wants to know more details about one violation, to take the correct decision, or if they need to configure new metrics to collect directly by the monitoring engine. Therefore, the interface will consist of both a REST interface and a publish/subscribe notification interface.

3.2.6 Manager

This component is connected to a queuing system, consumes all metrics coming from Prometheus beat, then publishes them to queues declared by others consumers component of the platform. Components of the platform need to subscribe in order to receive metrics in streaming mode. The subscription request requires the name of the component, the list of metrics to subscribe to, the name of the queue, a boolean parameter specifying if the value of the metric subscribed need to be different from the previous in order to be sent, a heartbeat interval. This last is very important because there is no way the publisher to know the connexion’s state of the consumer. Thus, the consumer should update its connexion’s state. Implementing a pub/sub mechanism without detecting the state of the consumer can easily lead to malfunctioning or waste of resources, the default value is 10 minutes. The subscription request is in fact a JSON content sent thought the queue “manager”. This is the format of the request.

```
{ "request": "subscription","name": "name_of_the_component","queue":
"queue_of_the_component","heartbeat_interval": 600,"Data":[{"name":"memory",
"labels":{"application":"web_service_1"},"on_change_only":true}]}
```

The above request will create a subscription to the metric memory coming from the application with the label “application” equals to “web_service_1”. The heartbeat interval is set to 600 seconds or 10 minutes. Because of the parameter “on_change_only”metrics will be published if only the current value of the current is different from the value of the previous one. Metrics are published in the queue “**queue_of_the_component**”.

```
{ "request": "subscription", "name": "name_of_the_component", "queue":
"queue_of_the_component", "heartbeat_interval": 600,
"data":[{"name":”*","labels":{"application":"web_service_1"},"on_change_only":true}  ]}
```

In the above example, the manager will publish all metrics generated by the application with the label “application” equal to “web_service_1”.

After having received the subscription request, the manager respond back in order to notify the status of the request then it starts sending into the queue declared metrics coming from the broker. The format of the response is:

```
{"status": "success", "data": {"labels": {"engine": "triple_monitoring_engine", "application":
"prometheusbeat", "job": "prometheusbeat", "component": "prometheusbeat", "instance":
"prometheusbeat:55673"}, "name": "datapoints_prometheus", "value": "1014", "time":
"2019-10-31T08:22:48.389000Z"}, "request": "stream"}
```

For each “heartbeat interval”, the consumer must send a heartbeat beat request to confirm the activeness of its connexion.

This is the JSON format of the heartbeat request.

```
 {“request”: “heartbeat”, “data”:
{“queue”:”queue_name”“name”:”name_of_the_component”}}
```

The monitoring engine provides also through the manager a functionality dedicated to the QoS Evaluator of the platform. This functionality is implemented in a streaming manner and provide a percentile of a bucket of metrics which tells the value at which a certain percentage of data is included. So a 95th percentile tells the value which is greater than or equal to 95% of your data.

The QoS Evaluator can guarantee the compliance of a SLO for the most part or a given period or time window. We define “for the most part” as the level of confidence we can have in the evaluation of the SLO.

There exist different ways in which we can “assess” a group of data points or measurements to determine whether they comply with the objective “for the most part”. One way is to aggregate data points in groups of n and determine whether the group as a whole complies with the objective. Again, there are different aggregation functions we can use: from quantiles/percentiles to mean (average) and median; we chose the former In other words, that, **metric’s value** is lower or higher than the objective for the **percentage** of measurements collected in the **time window**.

- *Response time < 900ms* for *99%* measurements collected in *10min*

This percentage can be calculated as the percentile 99th or 0.99 quantile (also known as 99% quantile), depending on the nomenclature we want to use. The formula is the following:

Index = (percentage)*(size+1)

Algorithm of percentile computation

Input: list_of_value, percentage

```
list_of_value.sort()
size = get_size_of_list(list_of_value)
index = percentage * (size + 1)
If index == size then
        Index = size - 1
return list_of_value[index]
```

The manager receives a "qos" request in order to start publishing the percentile to the corresponding queue of the QoS Evaluator. This request contains the name of the queue to reply to, the name of the request and a list where each element is an object composed by the name of metrics, the percentage, the name of the application producing the corresponding metric, the interval of time of the time window. This request has the following format:

```
{"request":"qos","queue":"qos",
"metrics":[{"application":"tester","metric":"scrape_duration_seconds","interval":10,"percentag
e":90}]}
```

The manager will be creating a bucket based on the interval of time specified in the request, then compute the percentile taking into account the percentage.

The output has the following format :

```
{"application": "tester", "metric": "scrape_duration_seconds", "percentile": "0.016867146",
"request": "qos"}
```

The computation is done as long as the manager does not receive a stop request.

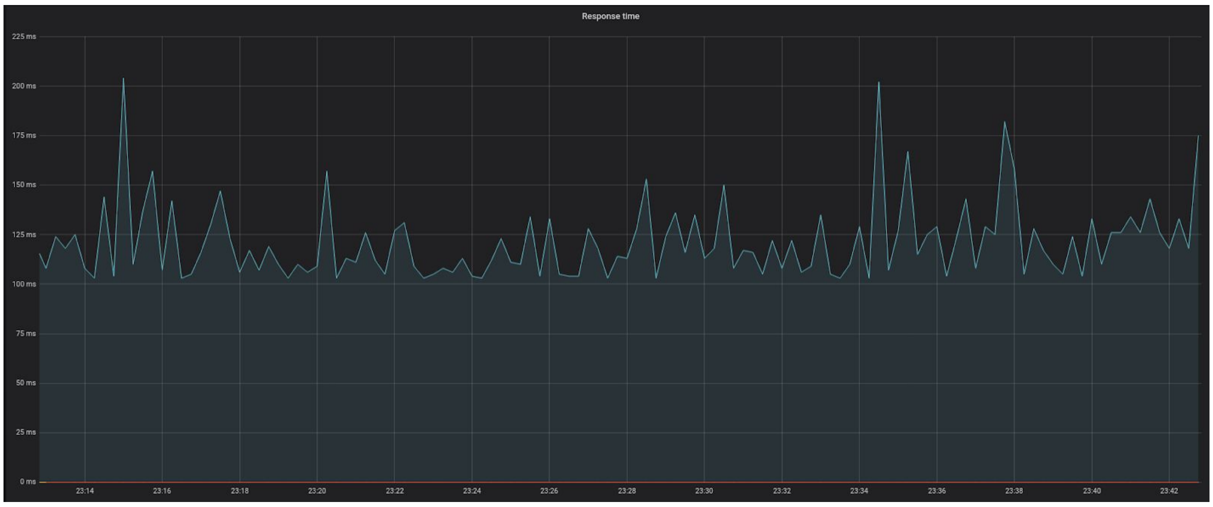
3.2.7 QoS Evaluator

The Qos Evaluator component is a key element of the monitoring since it evaluates an application agreement and compares it to the current value collected. The QoS Evaluator raises an alert in case of violation in order to inform the decision component for possible adaptation. This component receives as input the agreement which contains the application information, name of the metric, violation type and different threshold. It will then send a subscription request to the manager in order to start consuming the slo (metric) and evaluate the value with the different threshold defined in the agreement. After having started evaluating an application the QoS Evaluator will send a request to the predictor to his dedicated queue for allowing violation prediction.
The QoS Evaluator enables the platform to make decisions for satisfying user's requirements. The letter is defined in the agreement written as SLA (Service level agreement). In this document, it's described, application level evaluation element.
We provide below, a SLA model used to this platform.

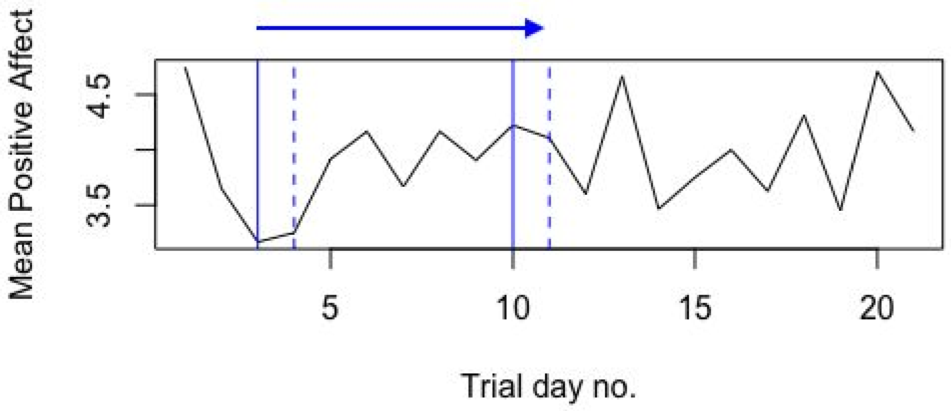
```
{
   "id": "a03",
   "name": "provide-recomendation-service",
   "state": "started",
   "details":{
       "id": "a03",
       "type": "agreement",
       "name": "provide-recomendation-service",
       "provider": { "id": "mf2c", "name": "mF2C Platform" },
       "client": { "id": "c02", "name": "A client" },
       "creation": "2019-01-16T17:09:45Z",
       "expiration": "2020-11-13T10:25:45Z",
       "messages":1,
       "frequency":60,
       "guarantees": [
           {
               "name": "responseTime",
               "constraint": "[responseTime] > 150",
               "importance": [
                       {
                               "Name": "Mild",
                               "Constraint": " > 150"
                       },
                       {
                               "Name": "Serious",
                               "Constraint": " > 200"
                       }
               ]
           }
       ]
   }
}
```

In the SLA, we have the application’s name, is and state. We describe the name of the metrics (SLO) which the key element from which we can measure the satisfaction of client’s requirements. With the SLO, there is the type of constraint “higher” or “lower” which define the threshold from which we can raise a violation. Violation can also be divided to different levels.

There are two main methods to evaluate a SLO. The first method is to compare each data point to the threshold. This method leads to unnecessary violation. This strategy is very to implement but leads to a bad management of resources since the platform can scale on an outlier.

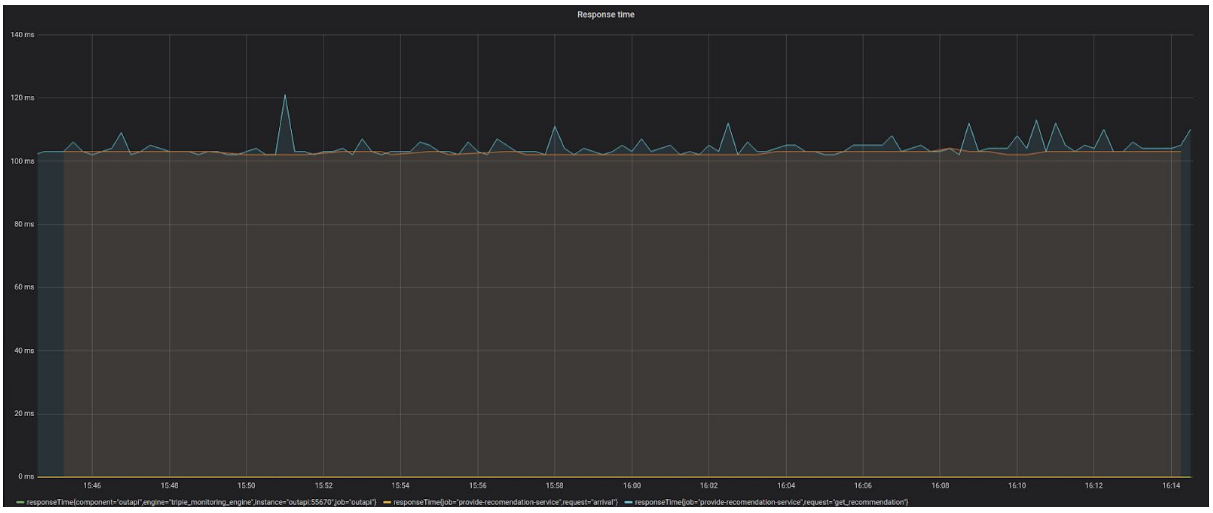


The above graph shows the evolution of the response time where the threshold is set to 175ms. We clearly observe that 3 data points reaches value higher than the threshold.
The next method is to evaluate the SLO by considering a set of data points. This introduces the notion of a time window which is a static interval of time a certain of metrics are collected.



From this notion, two methods come up, the first the find the average of all data point's value then compare this aggregation to the threshold. This method is inconvenient to mislead the evaluation. A single outlier can considerably modify the evaluation and lead to an unnecessary scale.

The method used in the QoS Evaluator is the percentile or median which a value below which a certain percentage of data points fall. Averages are ineffective because they are too simplistic and one-dimensional. Percentiles are a really great and easy way of understanding the real performance characteristics of your application. They also provide a great basis for automatic baselining, behavioral learning and optimizing your application with a proper focus.



Consider a data set of the following numbers: *122, 112, 114, 17, 118, 116, 111, 115, 112*.

N	Number
1	112
2	112
3	114
4	17
5	118
6	116
7	111
8	115
9	112

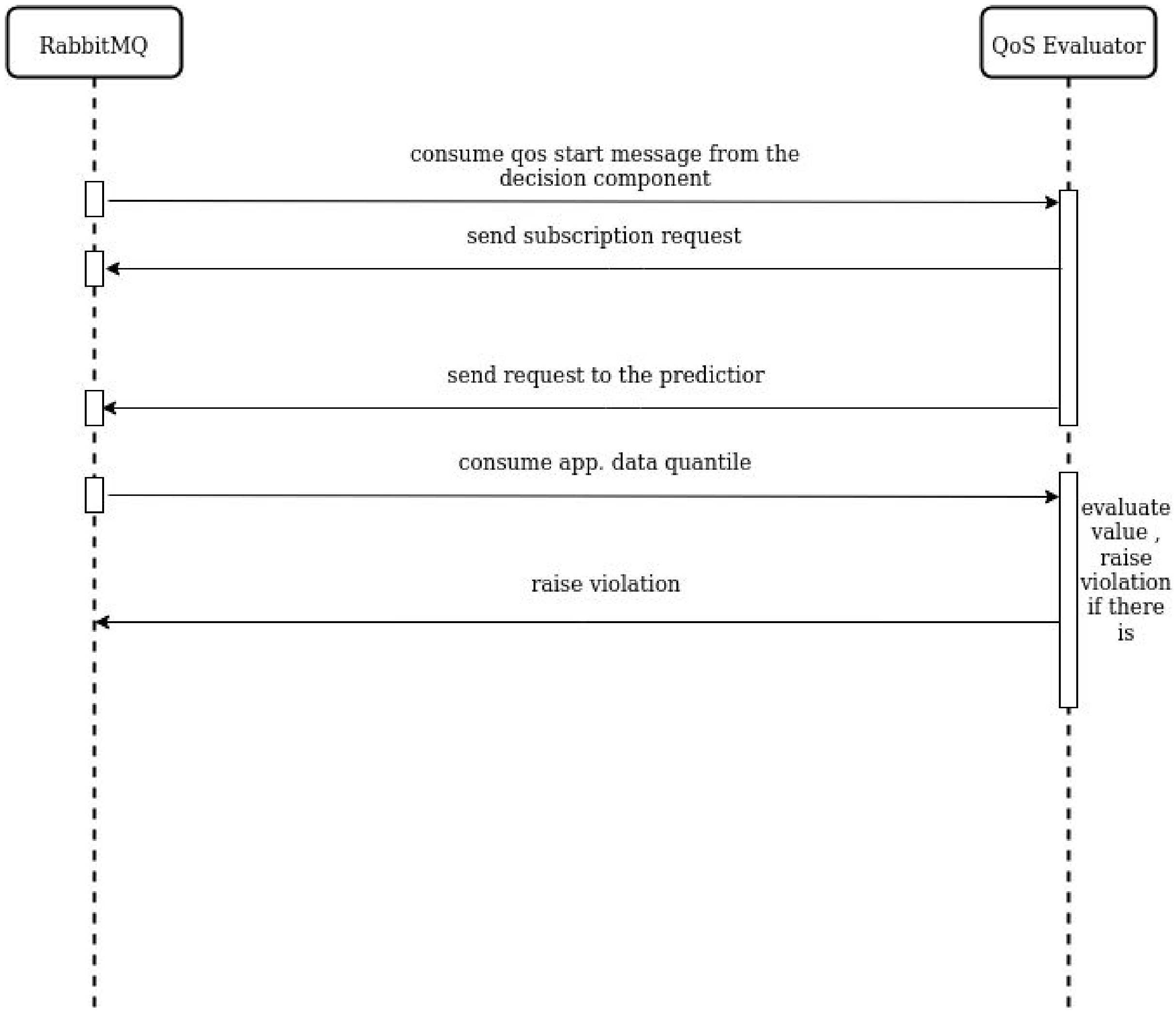
You are required to calculate the 25th Percentile Rank.
The first step is the order sample by ascending manner:
Data ordered : *17, 111, 112, 112, 114,115, 116,118, 122*

N	Number
1	17
2	111
3	112
4	112
5	114
6	115
7	116
8	118
9	122

The second is to find the rank which is given by the formula. R = (P/100)*(N+1). Where N is the number of data points and P the percentage.
The Rank is 2.5 so the concerned items are the second (111) and the third(112). The percentile will be the average of these two values (111+112)/2 = 111.5

To deliver a very good level of evaluation, thus to not miss any portion of time series which presents a violation pattern, a time window forward moving must be chosen. This factor expresses the number of steps we are moving on the time series. The window forward moving takes value between 1 and the window size. If it's very low, the QoS Evaluation .

Sequence diagram

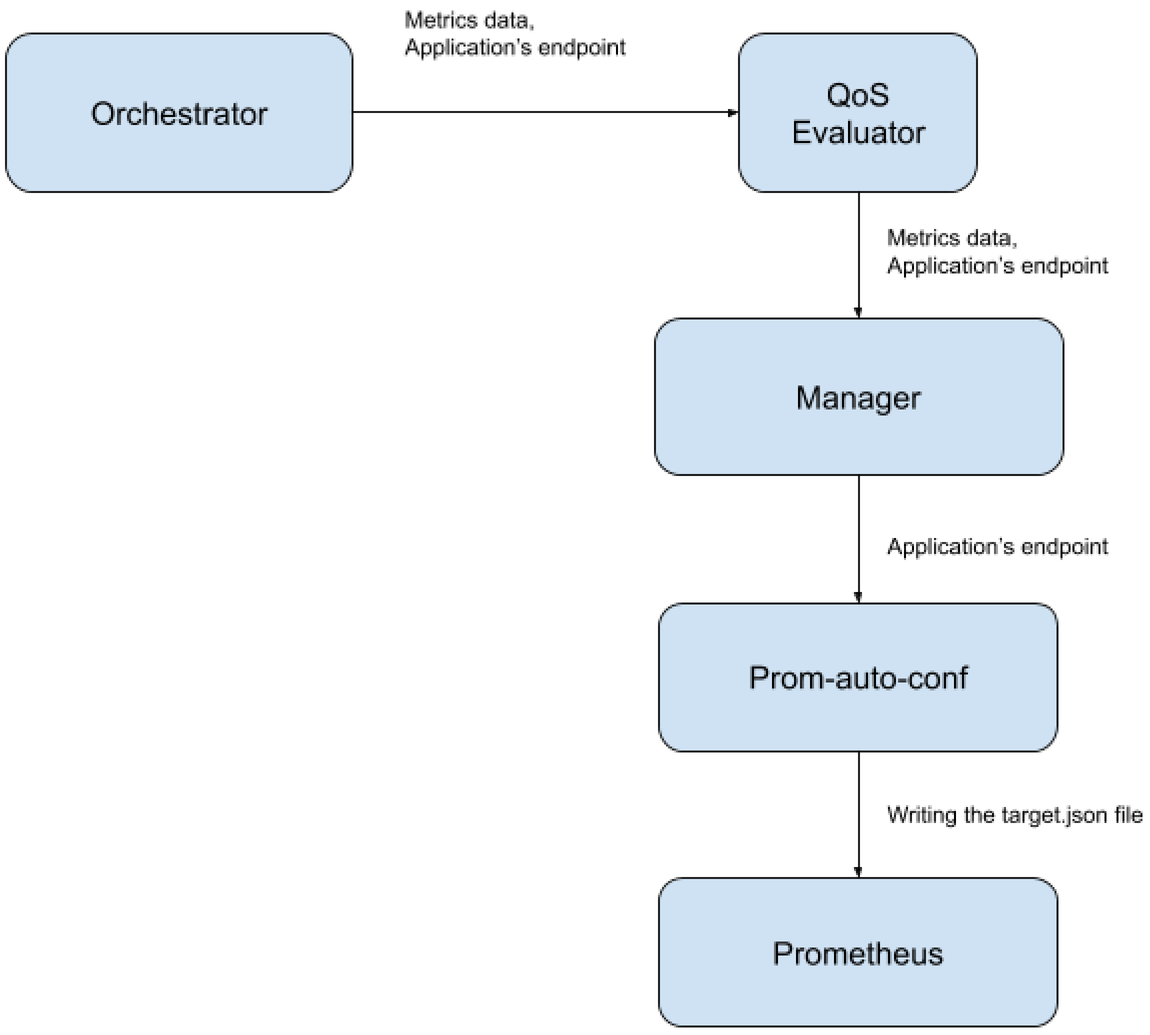

RabbitMQ
QoS Evaluator
consume qos start message from the decision component
send subscription request
send request to the predictior
consume app. data quantile
raise violation
evaluate value , raise violation if there is

The QoS evaluator is connected to the orchestrator of the platform where it receives the instruction to start monitoring an SLO and analysing it for detecting violation. The QoS Evaluator receives information related to metrics (SLO) to observe such:

- Metric's name
- Percentage, this information is used to compute the percentile
- Interval, the interval is converted to time window
- Application's name
- Component's name, the component's name is crucial where the application is composed of more than one component

- Replicas indice, since we are handling big data applications, it's important to manage replicas separately for better management.

QoS Evaluator receives also information related to application endpoint. We saw in the introduction the monitoring used by Prometheus. This latter needs to locate application in the network, thus knowing hostname or ip of the application and also the port the given application component is exposing metrics. Since those information is set by the orchestrator of the platform, the monitoring engine is receiving them from the QoS Evaluator.


Orchestrator
Metrics data, Application's endpoint
QoS Evaluator
Metrics data, Application's endpoint
Manager
Application's endpoint
Prom-auto-conf
Writing the target.json file
Prometheus

3.2.8 ML components

This component covers machine learning functionalities. The aim of this feature is to enable proactive violation alert systems. The monitoring through the QoS evaluator can notify the decision component about a violation. By reporting violation, a decision will be made. This later can be the increase or decrease of number of replicas, memory or cpu allocated. The change that will be made in the platform engages resources which are expensive and should be used efficiently. The QoS can report violation and reports the under utilization resource some moment after. This situation leads to unnecessary resource's allocation which leads to not efficient resource management. We would also like to raise the fact that, after the decision, the environment needs a certain moment in order to schedule actions and apply them. Both issues described can be addressed by providing the platform the prediction capability.

We would like to be able to detect violation before it happens. We won't fully implement these features on the current version however, we will analyse a SLO in order to find relevant influencers (metrics correlated with the given SLO) then, construct the dataset which will be used for prediction.

Detecting influencers or metrics correlated with the observed SLO is a time series operation which consists of evaluating the distance between two time series. Four approaches are on our disposal for time series correlation analysis.

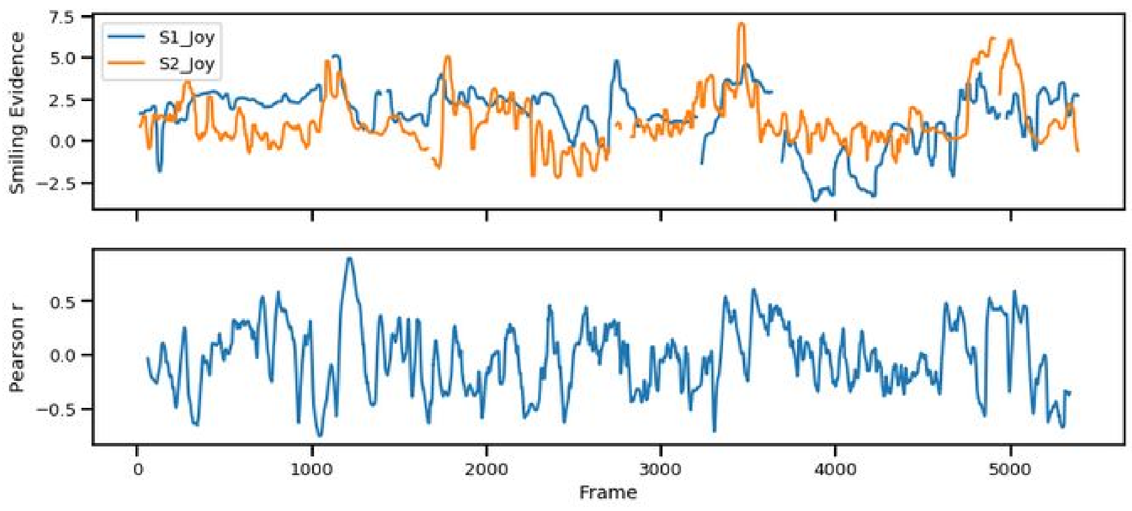
Pearson correlation

Pearson correlation is a measure of the linear correlation between two variables *X* and *Y*. Pearson correlation has a value between +1 and −1, where 1 is total positive linear correlation, 0 is no linear correlation, and −1 is total negative linear correlation Pearson correlation is computer as follow:

$$r = \frac{Cov(X,Y)}{\sigma x.\sigma y}$$

Where Cov(X,Y) is the covariance of the variable X and Y and

σx,σy are respectively the standard deviation of x and y.



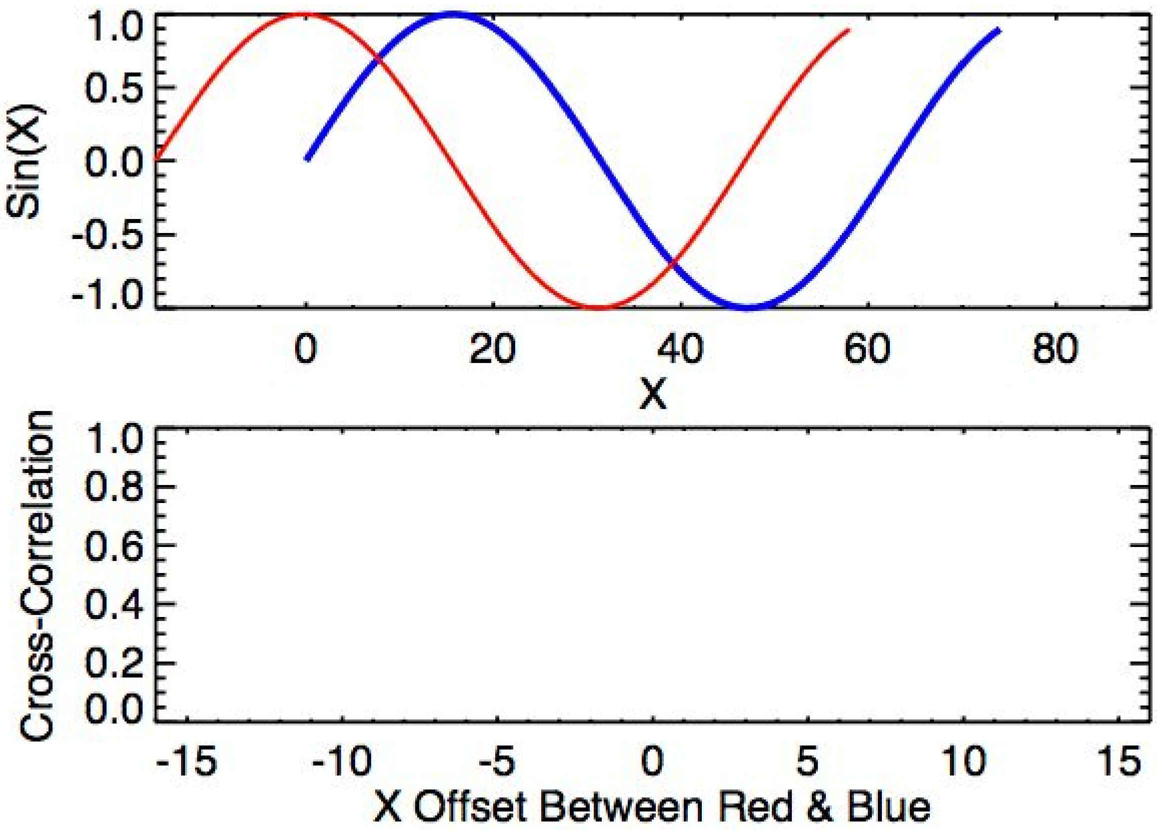
Two important things to take into account when using Pearson correlation is that:

1) outliers can significantly influence the correlation

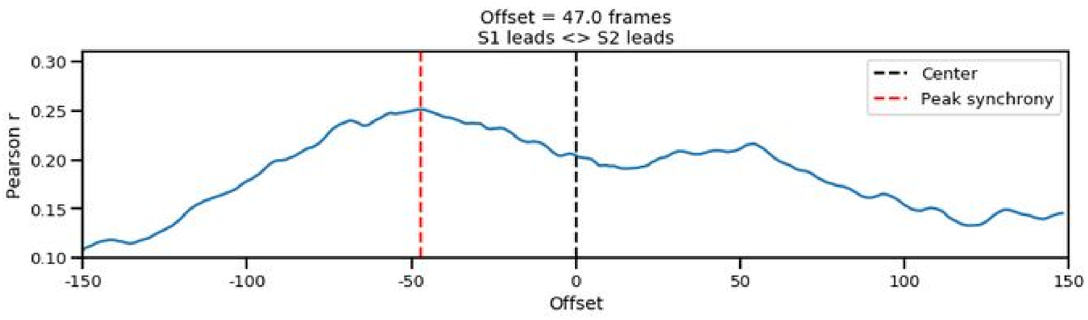
2) it assumes the data are homoscedastic such that the variance of your data is homogenous across the data range. Generally, the correlation is a snapshot measure of global synchrony. Therefore it does not provide information about directionality between the two signals such as which signal leads and which follows.

Time Lagged Cross Correlation

Time lagged cross correlation (TLCC) can identify directionality between two signals such as a leader-follower relationship in which the leader initiates a response which is repeated by the follower. There are a couple ways to investigate such relationships including Granger causality, used in Economics, but note that these still do not necessarily reflect true causality. Nonetheless we can still extract a sense of which signal occurs first by looking at cross correlations.



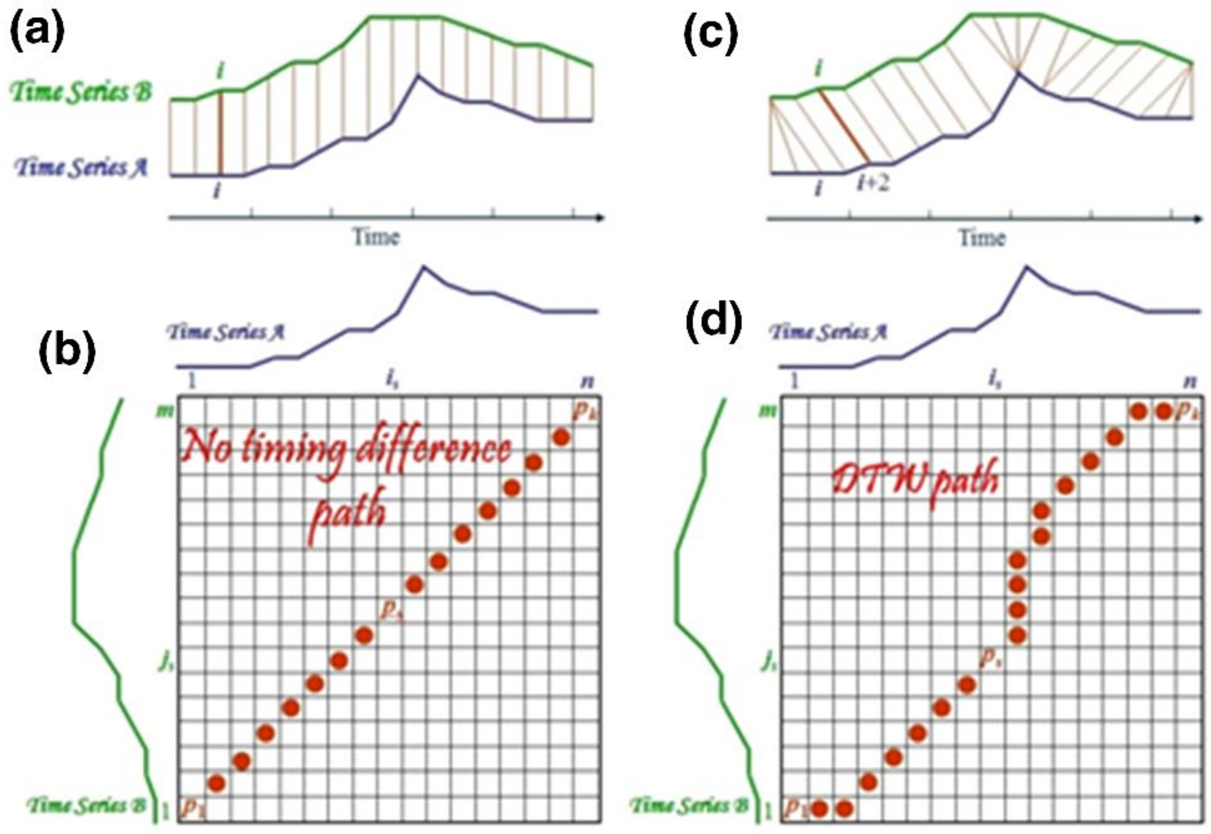
As shown above, TLCC is measured by incrementally shifting one time series vector (red) and repeatedly calculating the correlation between two signals. If the peak correlation is at the center (offset=0), this indicates the two time series are most synchronized at that time. However, the peak correlation may be at a different offset if one signal leads another. The code below implements a cross correlation function using pandas functionality. It can also *wrap* the data so that the correlation values on the edges are still calculated by adding the data from the other side of the signal. This method is very efficient for implementing bottleneck detection since we can determine which signal leads others.



In the plot above, we can infer from the negative offset that Subject 1 (S1) is leading the interaction (correlation is maximized when S2 is pulled forward by 47 frames). But once again this assesses signal dynamics at a global level, such as who is leading during the entire 3 minute period. On the other hand we might think that the interaction may be even *more* dynamic such that the leader follower roles vary from time to time.

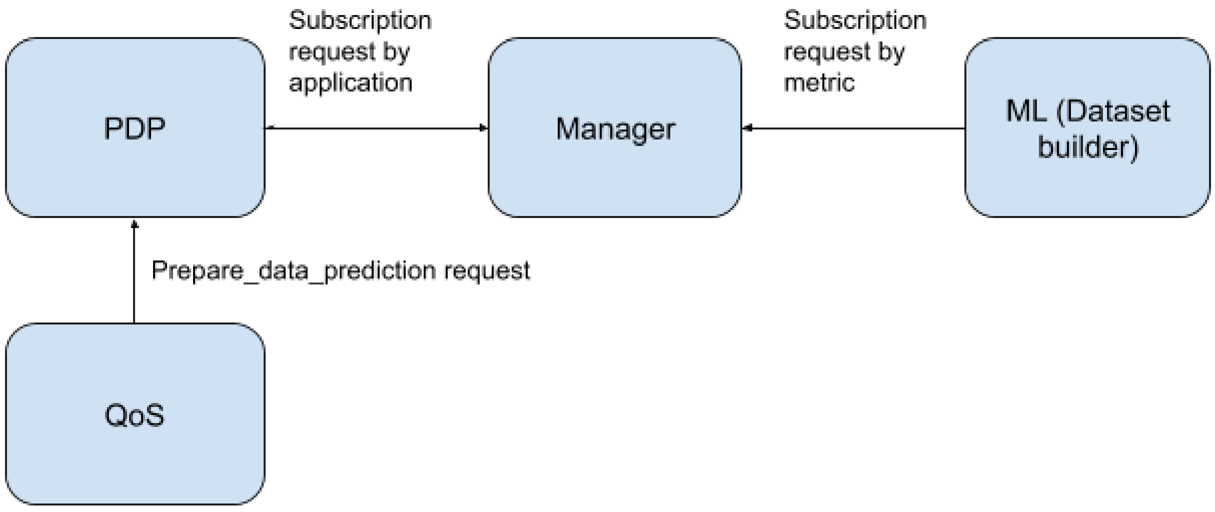
Dynamic Time Warping

Dynamic time warping (DTW) is a method that computes the path between two signals that minimize the distance between the two signals. The greatest advantage of this method is that it can also deal with signals of different length. Originally devised for speech analysis, DTW computes the euclidean distance at each frame across every other frame to compute the minimum path that will match the two signals. One downside is that it cannot deal with missing values so you would need to interpolate beforehand if you have missing data points.



For two given time series of n and m size, computing correlation with dynamic time warping method can be performed with the complexity of O(n*m) which is very good in terms of performance since we are dealing with time series of small size. The algorithm is the follow:

```
int DTWDistance(s: array [1..n], t: array [1..m]) {
    DTW := array [0..n, 0..m]
    for i := 1 to n
        for j := 1 to m
            DTW[i, j] := infinity
    DTW[0, 0] := 0
    for i := 1 to n
        for j := 1 to m
            cost := d(s[i], t[j])
            DTW[i, j] := cost + minimum(DTW[i-1, j  ],    // insertion
                                        DTW[i  , j-1],    // deletion
                                        DTW[i-1, j-1])    // match
    return DTW[n, m]
}
```


Subscription request by application
Subscription request by metric
PDP
Manager
ML (Dataset builder)
Prepare_data_prediction request
QoS

The QoS evaluator sends a prepare_data_prediction request after having started evaluating an SLO if the flag prediction is activated. The request is the follow:

```
{"request":"add_application","queue":"queue_name",
"data":{"name":
```

```
“application_name”,”slo”:”slo_metric_name”,”dependencies”:[app
lications        correlated        with        the        observed
one],”replicas”:”replicas_name”},”violation”:{“threshold”:”thr
eshold”,”threshold_type”:”>|<”,”under_utilization_thresold”:”l
imit”}}
```

The PDP receives the request then sends a subscription request to the manager. The PDP subscribes to all metrics created by the application specified in the “dependencies” field of the the prepare_data_prediction request. The PDP component starts to consume metrics then performing correlation by using dynamic time warping method for discovering correlation between time series. The result will be sorted using ascending method then send them with the threshold information to the ML component component for starting to build the dataset.

3.2.9 Optimizer

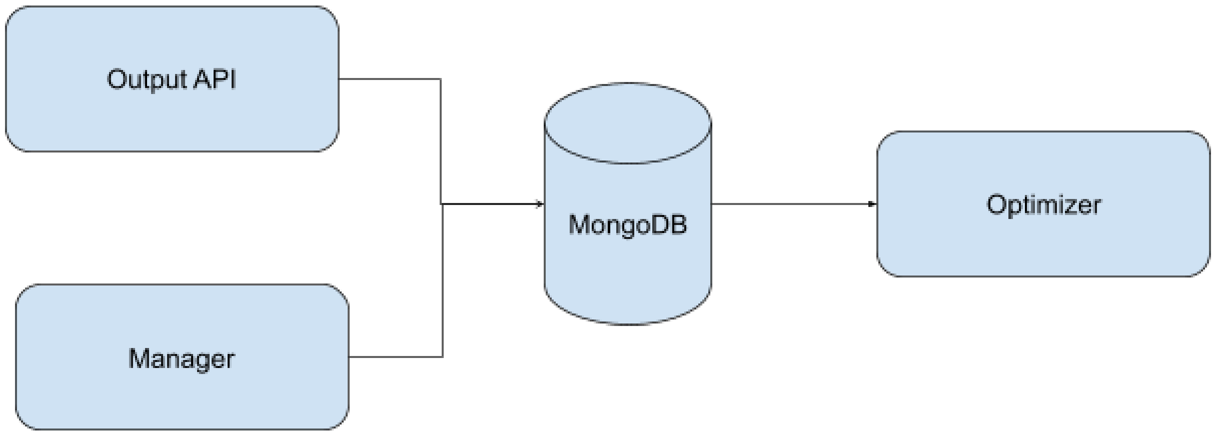
The goal of this component is to limit the amount of data points saved to the persistent repository. For allowing later analysis and for historical purpose, the platform needs to save metrics with their important related information. These information are:

- Name: the name of the metric
- Timestamp: the time, this metric has been collected
- Value: the value of the metric
- Labels: additional information of the metric

Before starting defining different strategies to avoid saving necessary metrics, we will analyze the amount of metrics produced by a single source (job). Basically the amount of metrics collected within an interval of time equals the number of sample produced by scrape multiplied by the number of scrape action within this interval of time. This can be expressed as follow:
For an interval of 2 minutes, or 120 seconds, a prometheus job that produced approximately 15 samples by scrape where the scrape interval is 4 seconds, the number of metrics collected will be equal to (120/5)*15 = 360 data points. We can understand that the amount of metrics produced is very related to the scraping interval. The scraping interval is defined at configuration and should be adjusted in order to provide to the metrics consumer enough information. Two situations can happen, the first one is the situation where the scraping interval is higher than the ideal one, this can lead to some malfunctioning of the decision component since it can miss some important event. The second situation is the scraping interval is lower than the ideal one. This situation leads the collector to gather data points that will be used. It’s very difficult to define the ideal scraping interval since it involves two entities which are not necessarily together.

The first approach is to define the ideal scraping interval by observing the utility of job, this is defined by finding the maximum frequency a metric from a given job is requested.
For a job where 10 metrics, if the maximum request rate of metrics from this given job is 5 seconds, the scraping will be dynamically modified to 5. This approach guarantees us to set scraping in such a way to not lose any important event but this is not the correct response to our challenge to not save unnecessary data points. The above approach has an effect on an entire job which contains more than one metric. The second approach is to handle each metric separately. We will be defining the utility of a metrics (how often this metric is requested) and the saving interval will be adjusted accordingly.
The implementation implies the monitoring to have the capability to track the request on metrics that saved these tracking information into a database. These tracking information will be used by the optimizer for adjusting the scraping interval of each job and also adjust the saving interval.
Since prometheus is our official collector, after having modified its configuration file, there is a need to send a restart request for reloading the new configuration. In order to avoid many restart a level of acceptance must be set. This later will compared to the monitoring scraping performance which can be expressed as follow:

Scraping performance = (ideal_scraping_all_jobs/current_scraping_all_jobs)*100



On the above graph, we have the architecture for implementing the optimization at the monitoring engine. The output and the manager save to MongoDB element related to the metric and the moment of access. The json file send to MongoDB is the follow:

```
{'index':’unique_metric_index’,'type':'stream      |      ad-hoc','last':
‘time_of_access’,'name':
name_of_the_metric,'n_access':0,'from':int(time.time()),'job':
_job_name}
```

Index: is the hash of the name of the metric combined with the metric job's name. In case there is no job's name specified in the request, the name "all" is replaced.
Type: the output api uses the value 'ad-hoc' and the manager uses the value 'stream'
'Name': name of the metric
'N_access': the number of access. This value is incremented if the same index is found in the database.
'From': is the first time, the given index has been requested
'Last': is the last time , the given index has been requested. This value is updated if the given index is found in the database.
The optimizer reads first Prometheus' configuration file, to load the current scraping interval of all jobs. The optimizer loads periodically tracking information stored in MongoDB in order to compute the frequency each metrics saved is requested in order to compute the scraping performance indicator. The loading interval is defined to the optimizer's configuration file. The default value is set to 4 minutes.

3. Result

We will develop in this chapter, different methods and techniques used for getting result that justify the efficiency of the approach adopted in this project. We have to main Key Indicator Performance (SLO) of the monitoring engine.
The execution environment is Core i7 (8th Generation) computer with 8G of memory. The entire system is built on container technology with docker. Here is the docker-compose file.

```
version: '2'
services:
 prometheus:
   image: prom/prometheus
   hostname : prometheus
   restart: always
   container_name: prometheus
   networks:
     - monitoring
   ports:
     - 9090:9090
   volumes:
     - "./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml:z"
     - "./prometheus/targets.json:/etc/prometheus/targets.json:z"
   command:
     - '--config.file=/etc/prometheus/prometheus.yml'
```

```
      - '--web.console.libraries=/etc/prometheus/console_libraries'
      - '--web.console.templates=/etc/prometheus/consoles'
      - '--storage.tsdb.retention.time=52h'
      - '--web.enable-lifecycle'

  pushgateway:
    image: prom/pushgateway
    hostname: pushgateway
    restart: always
    container_name: pushgateway
    networks:
      - monitoring
    ports:
      - 9091:9091

  prometheusbeat:
    image: jdtotow/prometheusbeat
    hostname: prometheusbeat
    restart: always
    container_name: prometheusbeat
    networks:
      - monitoring
    ports:
      - 55673:55673
    environment:
      - "PROMETHEUS_URL_API=http://prometheus:9090"
      - "EXPORTER_URL=http://localhost:55673"
      -
"METRICS_SOURCE=manager,qos,prometheusbeat,qos_quantile,prometheus,prom
etheusbeat2,outapi,node_exporter,rabbitmq,pushgateway,exporter,bigdatas
tack-users-apps"
      - "RABBITMQ_HOST=rabbitmq"
      - "SLEEP=0.1"
      - "COMPONENTNAME=prometheusbeat"
      - "UPDATEMETRICSLISTNAMEPERIOD=30"
      #- "HTTP_OUT_URL=http://logstash:8089"

  prometheusbeat2:
    image: jdtotow/prometheusbeat
    hostname: prometheusbeat2
```

```yaml
   restart: always
   container_name: prometheusbeat2
   networks:
     - monitoring
   ports:
     - 55679:55679
   environment:
     - "PROMETHEUS_URL_API=http://prometheus:9090"
     - "EXPORTER_URL=http://localhost:55679"
     -
"METRICS_SOURCE=manager,qos,prometheusbeat,qos_quantile,prometheus,prom
etheusbeat3,prometheusbeat2,outapi,node_exporter,rabbitmq,pushgateway,e
xporter,bigdatastack-users-apps"
     - "RABBITMQ_HOST=rabbitmq"
     - "SLEEP=0.1"
     - "COMPONENTNAME=prometheusbeat-2"
     - "UPDATEMETRICSLISTNAMEPERIOD=32"
     #- "HTTP_OUT_URL=http://logstash:8089"
     - "EXPORTERPORT=55679"
     - "DEPLOYMENT=secondary"
 prometheusbeat3:
   image: jdtotow/prometheusbeat
   hostname: prometheusbeat3
   restart: always
   container_name: prometheusbeat3
   networks:
     - monitoring
   ports:
     - 55689:55689
   environment:
     - "PROMETHEUS_URL_API=http://prometheus:9090"
     - "EXPORTER_URL=http://localhost:55689"
     -
"METRICS_SOURCE=manager,qos,prometheusbeat,qos_quantile,prometheus,prom
etheusbeat3,prometheusbeat2,outapi,node_exporter,rabbitmq,pushgateway,e
xporter,bigdatastack-users-apps"
     - "RABBITMQ_HOST=rabbitmq"
     - "SLEEP=0.1"
     - "COMPONENTNAME=prometheusbeat-2"
     - "UPDATEMETRICSLISTNAMEPERIOD=32"
```

```
    #- "HTTP_OUT_URL=http://logstash:8089"
    - "EXPORTERPORT=55689"
    - "DEPLOYMENT=third"
 #exporter:
#  image: jdtotow/exporter
#  container_name: exporter
#  hostname: exporter
#  restart: always
#  networks:
#    - monitoring
#  ports:
#    - 55671:55671
outapi:
  image: jdtotow/output:dci
  container_name: outapi
  hostname: outapi
  restart: always
  networks:
    - monitoring
  ports:
    - 55670:55670
  environment:
    - "ELASTICSEARCHHOST=elasticsearch"
    - "MONGODBHOST=mongodb"
    - "PROMETHEUS_URL_API=http://prometheus:9090"
    - "PROCESSINGDELAY=120"
    - "DEFAULTEND=30"
rabbitmq-exporter:
  image: kbudde/rabbitmq-exporter
  container_name: rabbitmq-exporter
  hostname: rabbitmq-exporter
  restart: always
  networks:
    - monitoring
  ports:
    - 9419:9419
  environment:
    - "RABBIT_URL=http://rabbitmq:15672"
    - "RABBIT_USER=richardm"
    - "RABBIT_PASSWORD=bigdatastack"
```

```
#elasticsearch:
#  image: docker.elastic.co/elasticsearch/elasticsearch:6.6.1
#  container_name: elasticsearch
#  hostname: elasticsearch
#  restart: always
#  networks:
#    - monitoring
#  ports:
#    - 9200:9200
#    - 9300:9300

#logstash:
#  image: docker.elastic.co/logstash/logstash:6.4.3
#  container_name: logstash
#  hostname: logstash
#  restart: always
#  networks:
#    - monitoring
#  environment:
#    - "ELASTICSEARCHHOST=elasticsearch"
#    - "LOGSTASHHOST=logstash"
#  volumes:
#    -
"./logstash/logstash.conf:/usr/share/logstash/pipeline/logstash.conf"
#  ports:
#    - 8089:8089

qos:
  image: jdtotow/qos
  hostname: qos
  restart: always
  container_name: qos
  networks:
    - monitoring
  ports:
    - 55682:55682
  environment:
    - "RABBITMQHOSTNAME=rabbitmq"
    - "CONFIGFILEPATH=/config"
    - "EXPORTERPORT=55682"
```

```
      - "EXPORTER_URL=http://qos:55682"
  pdp:
    image: jdtotow/pdp
    hostname: pdp
    restart: always
    container_name: pdp
    networks:
      - monitoring
    environment:
      - "RABBITMQHOSTNAME=rabbitmq"
      - "CONFIGFILEPATH=/config"
      - "EVALUATIONINTERVAL=600"
 ml:
    image: jdtotow/ml
    hostname: ml
    restart: always
    container_name: ml
    networks:
      - monitoring
    volumes:
      -
"/home/jean-didier/Projects/bigdatastack/TME/ml/src/dataset:/dataset"
    environment:
      - "RABBITMQHOST=rabbitmq"

 recommender:
    image: jdtotow/recommender
    hostname: recommender
    container_name: recommender
    networks:
      - monitoring
    restart: always
    environment:
      - "PUSHGATEWAY=pushgateway:9091"
    ports:
      - 7070:7070

 manager:
    image: jdtotow/manager
    hostname: manager
    restart: always
    container_name: manager
    networks:
      - monitoring
    ports:
      - 55671:55671
      - 55683:55683
    environment:
      - "MONGODB_HOST=mongodb"
      - "RABBITMQHOST=rabbitmq"
      - "URLEXPORTER=http://localhost:55671"
      - "COMPONENTNAME=manager"
      - "NTHREADSCONSUMER=5"
      - "URLEXPORTERQOS=http://localhost:55683"

  rabbitmq:
    build: rabbitmq/.
    hostname: rabbitmq
    container_name: rabbitmq
    networks:
      - monitoring
    ports:
      - 5672:5672
      - 5671:5671
      - 15672:15672

  mongodb:
    image: mongo
    hostname: mongodb
    container_name: mongodb
    restart: always
    environment:
      - "MONGO_INITDB_ROOT_USERNAME=uprc"
      - "MONGO_INITDB_ROOT_PASSWORD=bigdatastack"
      - "MONGO_INITDB_DATABASE=TPME"
    networks:
      - monitoring
    ports:
      - 27017:27017
  grafana:
```

```yaml
   image: grafana/grafana
   hostname: grafana
   container_name: grafana
   restart: always
   networks:
     - monitoring
   ports:
     - 3000:3000

networks:
 monitoring:
   driver: "bridge"
```

1. Monitoring freshness or monitoring delay

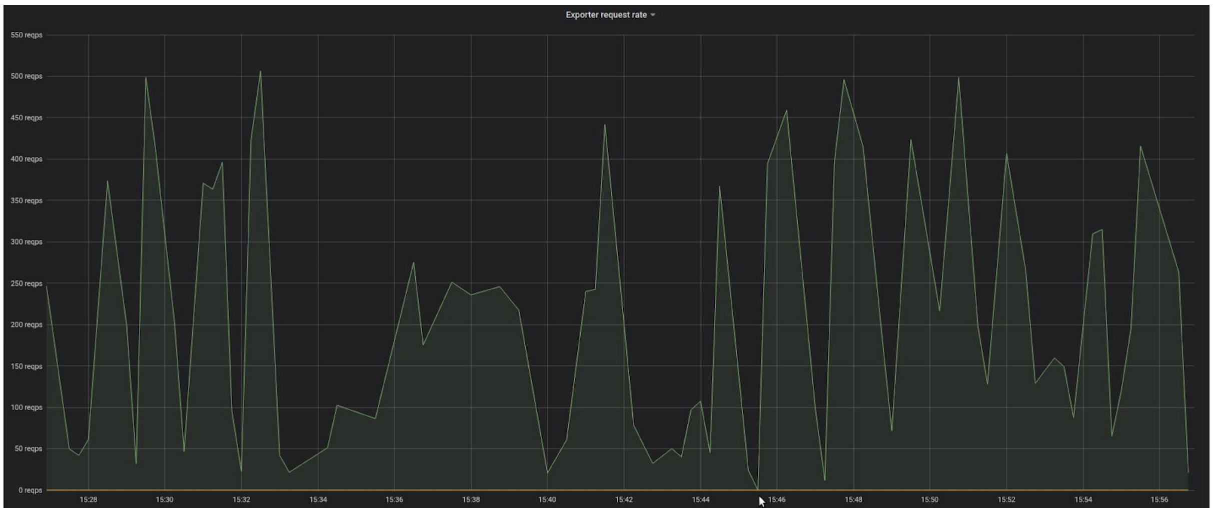
In order to deliver a prompt adaptation, the data-driven environment needs to react as fast as possible when an event occurs. This requirement implies that the monitoring engine needs to have a very slow delay. Thus we will perform many cases where we will be modifying the number of metrics collected and measure the delay of the engine. We will also observe the impact of the number of PrometheusBeat's instances on the entire engine.

PrometheusBeat instance = 1

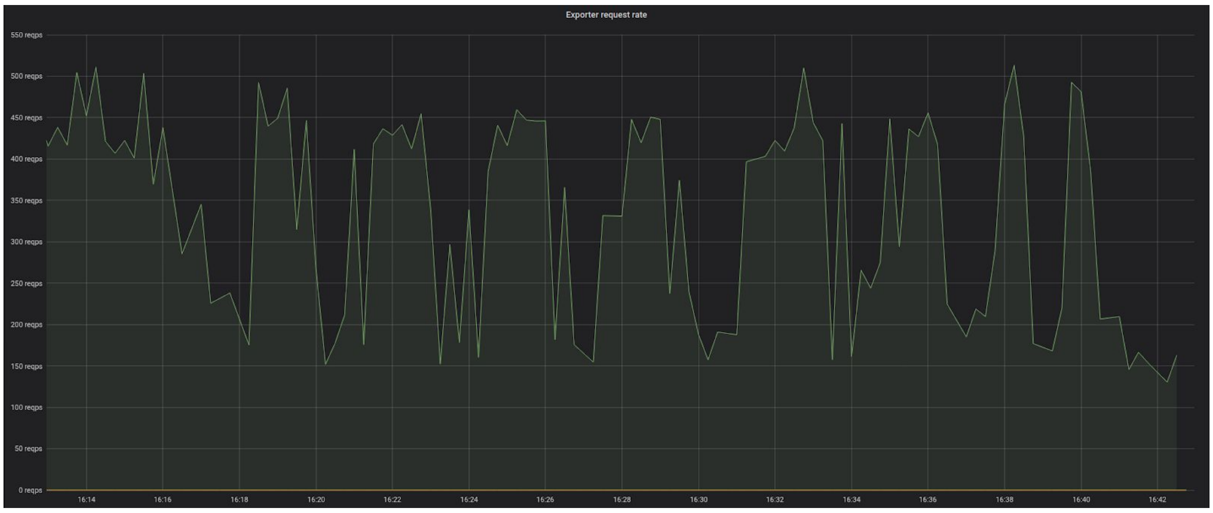
Max metrics number = 2600

Metrics consumed rate, this is the indicator of the rate the manager consumes metrics

Component: Manager

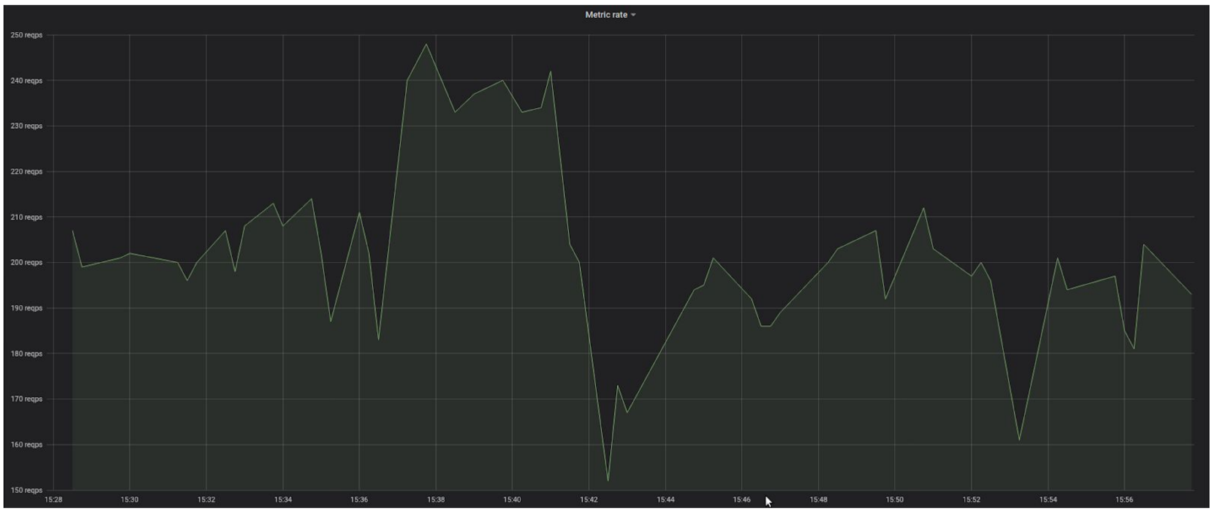


PrometheusBeat Instance = 2

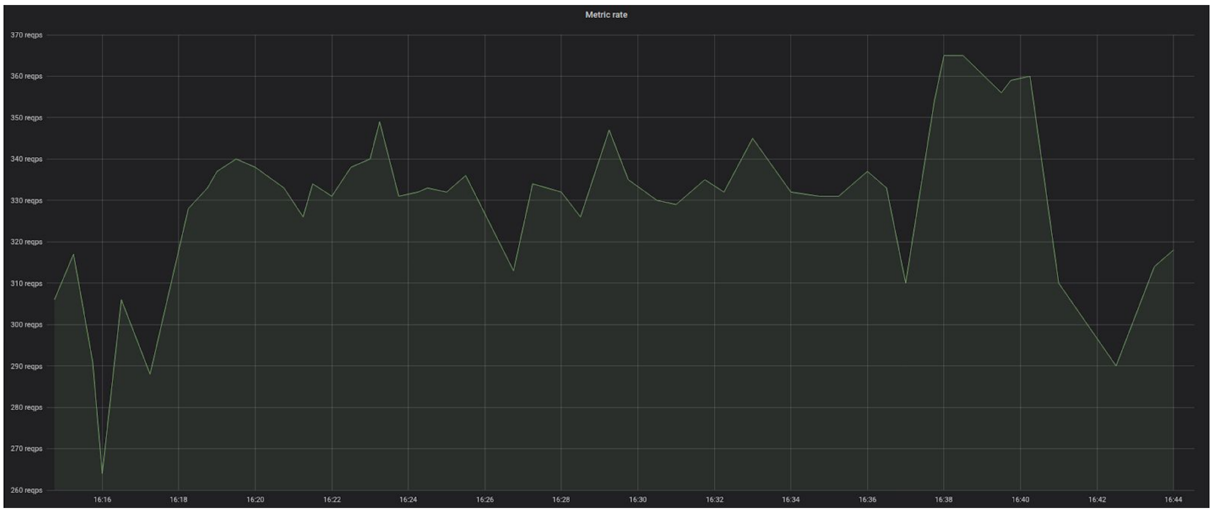


This indicator determines the performance of the manager, in others words, it defines the speed the manager is handling metrics. Since the Manager is the center of the monitoring engine, the quicker it consumes metrics the quicker the component will be. This indicator is affected by the number of PrometheusBeat instances.

Metrics exported rate
PrometheusBeat instance = 1
Component: PrometheusBeat

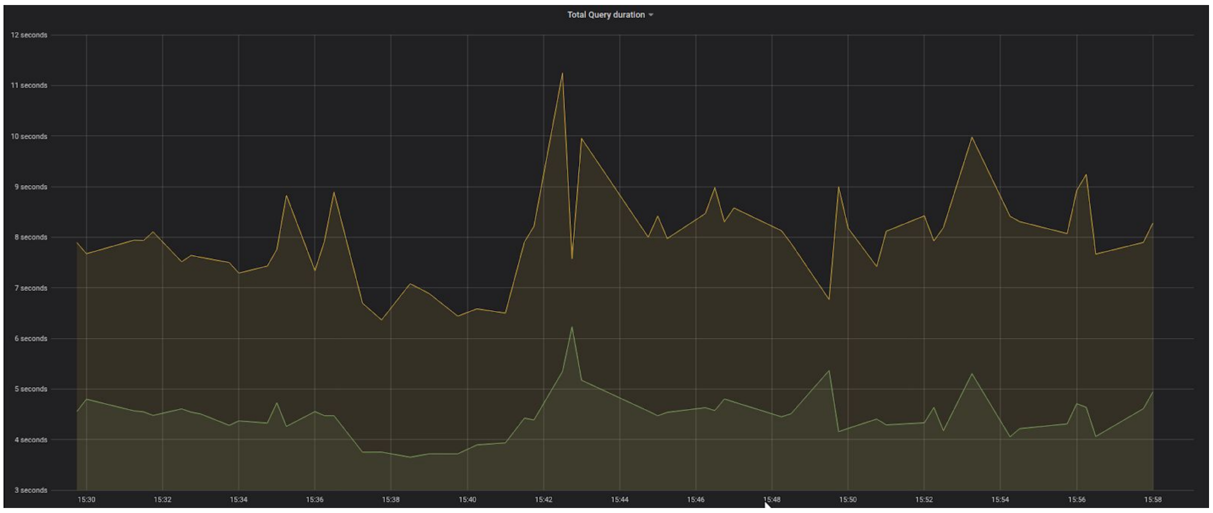


PrometheusBeat instance = 2

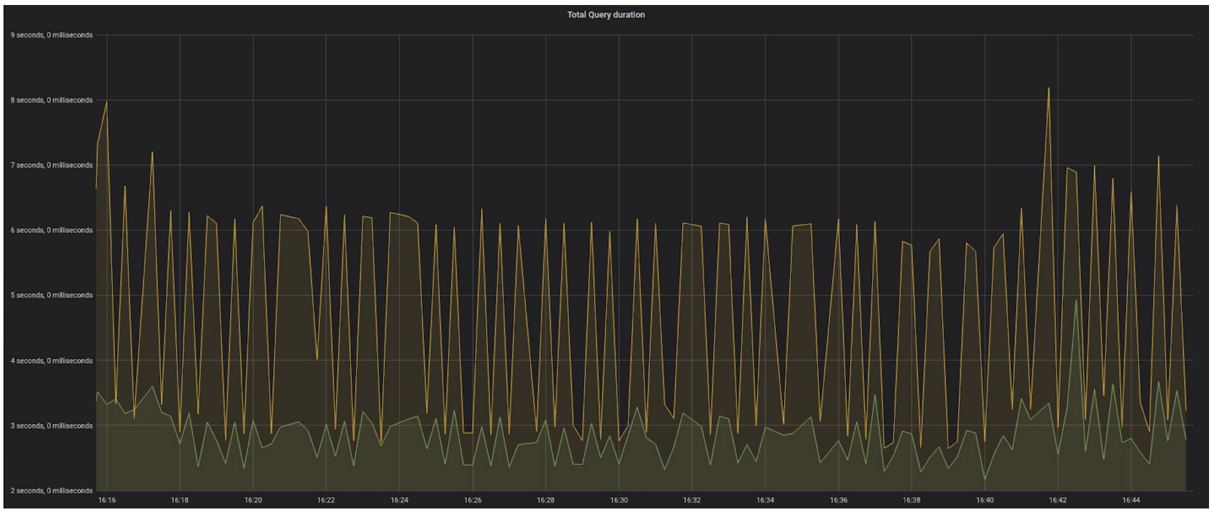


The export rate is the measurement that allows us to determine the speed, PrometheusBeat components publish metrics to the Manager. This metric is influenced by the number of metrics read from Prometheus.

Monitoring Latency : PrometheusBeat Instance = 1



Monitoring Latency: PrometheusBeat Instance = 2

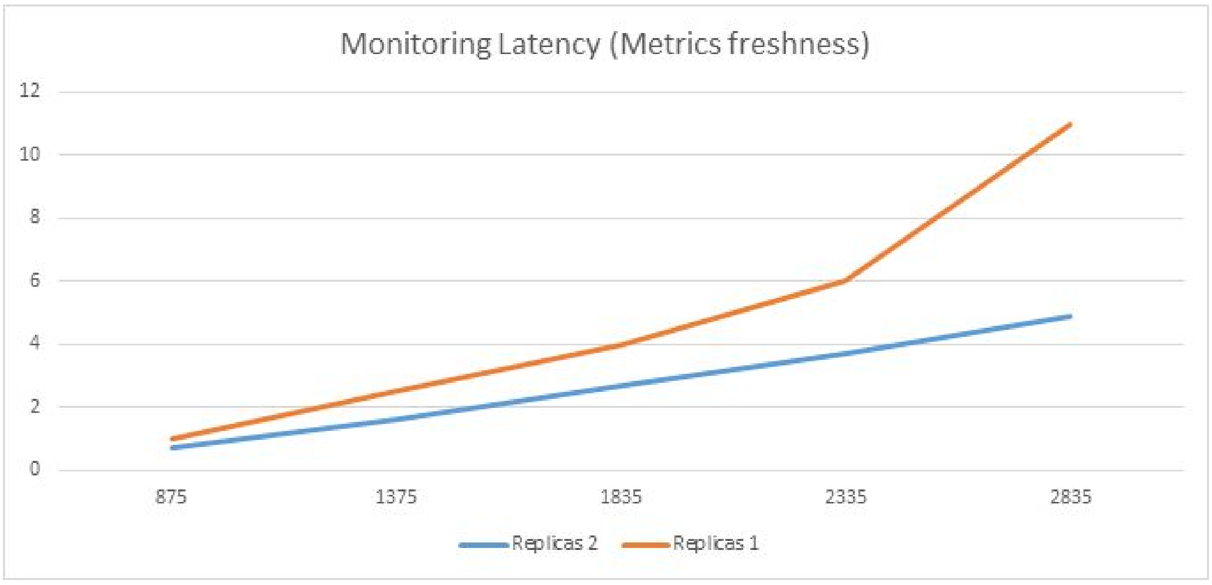


One of the goals of this project is to minimize the monitoring latency in order to allow a fast response to violation. We can observe two time series on the two graphs above. The first in green is the query time. The time needed by PrometheusBeat to retrieve metrics from Prometheus API. The second time series in yellow is the time required by the PrometheusBeat for publishing metrics. The Latency of the monitoring engine is the sum of these two values. And the latency is proportional to the number of metrics collected by prometheus.

This indicator is influenced by the the number of metrics exported and also by the performance of Prometheus.

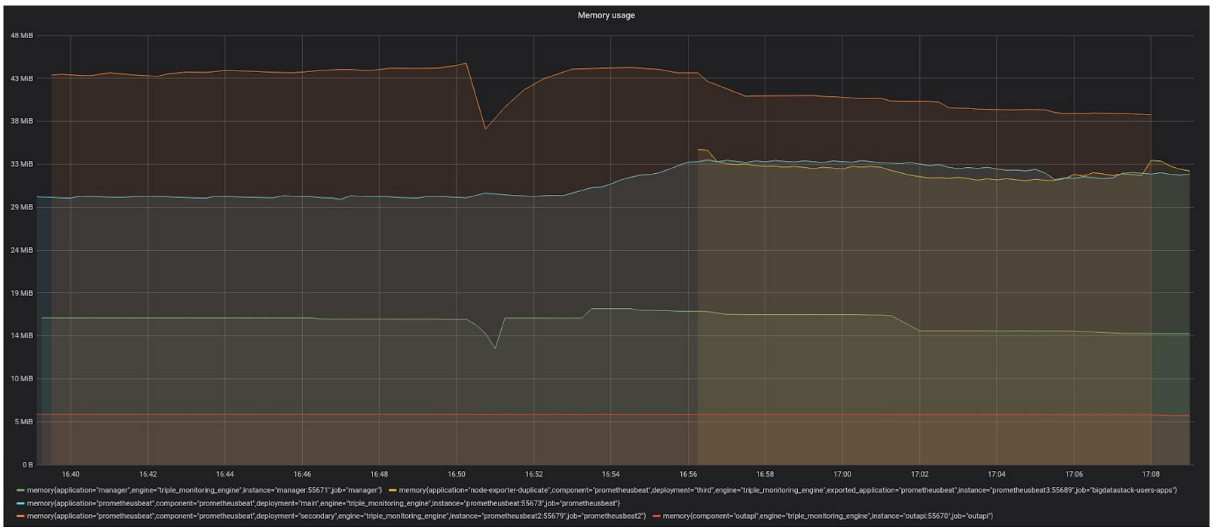
The increase of PrometheusBeat instances show the fall of this indicator.

In the below graph we will observe the evolution of the monitoring latency based on the amount of metrics collected.



2. Memory consumption

The volume of data is increasing over the year and the resources that have to be allocated costs proportionally with the amount of data. One the indicator of performance of the monitoring engine will be the lowest resource utilization as possible with a huge amount of metrics handled. We will correlate the memory consumption with the number of metrics collected.
In order to apply the strategy described above, we will be using Grafana’s dashboard for visualization.



In this graph we can observe the evolution of memory consumption of different components of the monitoring engine. All components consume less than 50Mb which is very encouraging in a big data environment.

3. Storing performance

After having applied the scraping optimization and the data points selection strategy. The amount skipped equaled the total number of metrics in the platform minus the amount of metrics used. The monitoring engine has a performance indicator of at least 90%.
This indicator is computed by :
Performance = (default_scrape_unused - old_scraping_interval) / default_scrape_unused
Unused data points saved = number_unused_metrics * (handling_time_window/ default_scrape_unused)

Conclusion

The monitoring engine is crucial in a data-driven environment since implements mechanisms to report the state and performance indicators of applications and of the platform by providing in real time metrics. The evaluation of those measurements are performed by comparing an objective value with the preference defined by the end user in order to deliver a better client experience. The monitoring implements also methods for storing data points for historical purposes. When the amount of metrics are important for delivering details about applications involved in the system performance, it also introduces a significant latency which can lead to delay at the response to violation and important event. This trade-off has been addressed by applying distribution technique of the component ingesting metrics to the queuing system of the engine. Two methods of applying distribution have been studied. The first one consists of dispatching tasks by separating metrics source amongst Prometheus Beat workers. We demonstrated that metrics sources don't necessarily have the same amount of metrics, therefore this approach creates an imbalanced distribution of tasks amongst workers which does reduce significantly the latency of the monitoring engine. The second approach and the approved one was the distribution of metrics name amongst workers, this technique divides the publication time which is the main factor of the latency to the number of workers. However , the query time linked to the monitoring collector stayed unchanged. We showed that the latency affects the amount of metrics the monitoring can handle.
The second challenge concerns the storage performance. The collector queries exporter for receiving metrics after some interval of time defined to the configuration file of the collector (prometheus). This interval is the main factor of the size of the prometheus. This interval of time or scrape interval is not necessarily defined based on the consumption rate. Therefore, we are collecting and saving metrics who won't be used or they won't be requested at the rate they are being saved. The approach developed to address this challenge is to match the consumption rate with the collection rate then reload the prometheus for applying changes. This technique allows the monitoring to reduce the amount of metrics collected, thus reducing the monitoring latency and the storing space.

Future work

Improving monitoring latency: The approach used in this project exploits the reduction the time expended for publishing metrics to the queuing system where another part of the latency is created from the querying time to Prometheus. We aim to reduce the latency by using more than one Prometheus instance. Jobs will be distributed amongst Prometheus instances.

Proactive alert: The current monitoring engine via the QoS Evaluator, can detect a violation of the agreement then send an alert message to a decision component (orchestrator). The orchestrator will then elaborate a set of actions for applying suitable changes. Computing changes for a reconfiguration has a time cost which can be critical for real-time applications. We plan to address this challenge by completing the ML component we started developing in this project. The ML after having created a dataset, will train it and start prediction for delivering a proactive behavior in the monitoring engine.

Universal exporter: Prometheus requires the hostname of IP in order for requesting metrics from an exporter. Applications monitored in the platform need to have an IP or a hostname. Since one of the actions the orchestrator can take is the duplication of the application (scale up), the platform needs to provide an IP or hostname to the new born replicas. IP and hostname are expensive in an environment where an application by its requirements and load can lead to more that 100 replicas. We will address this situation by creating an exporter which will listen to a specific queue then exporter metrics published to that queue to Prometheus. The application running on the platform will either implement a queueing system client or an http client since a pipeline can be established presenting an endpoint to application developers and publishing to queueing system.

Bottleneck detection: The ML component using time series correlation technique can find relevant influencers metrics of a given SLO. The correlation method used does not classify the metrics leader. We would like to improve this component by implementing a set of methods for detecting the metrics leader in order to deliver a full completed report to the orchestrator for a better quality of service. The orchestrator will be able to improve the performance of the application not only by applying modification but also by improving the performance of all applications influencing the monitored one.

Bibliography

[1] BigDataStack,bigdatastack , www.bigdatastack.eu, [01/10/2018]
[2] Prometheus, prometheus, www.prometheus.io , [04/10/2018]
[3]Percentile, Elastic,
https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-metrics-percentile-aggregation.html
[4] Time series correlation,
https://towardsdatascience.com/four-ways-to-quantify-synchrony-between-time-series-data-b99136c4a9c9
[5] Time series correlation, Pearson method,
https://www.spss-tutorials.com/pearson-correlation-coefficient/
[6] QoS Metrics for Cloud Computing Services Evaluation, Amid Khatibi Bardsiri
Computer Engineering Department, Bardsir Branch Seyyed Mohsen Hashemi
Assistant Professor, Science and Research Branch, Islamic Azad University, Tehran, Iran
[7] Spark measure,
https://github.com/LucaCanali/sparkMeasure/blob/master/docs/prometheus.md
[8] Time series theory, https://en.wikipedia.org/wiki/Time_series
[9] Cloud Native Monitoring with Prometheus,
https://samirbehara.com/2019/05/30/cloud-native-monitoring-with-prometheus
[10] Monitoring for cloud environment, Melodic, https://melodic.cloud
[11] An Ontology-driven Approach to Self-management in Cloud Application Platforms, Rustem Dautov, Iraklis Paraskakis, Dimitrios Kourtesis
[12] Shroff, G.: Enterprise Cloud Computing: Technology, Architecture, Applications. Cambridge University Press (2010).
[13] Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: Above the Clouds: A Berkeley View of Cloud Computing, (2009).

[14] Stojanovic, L., Schneider, J., Maedche, A., Libischer, S., Studer, R., Lumpp, T., Abecker, A., Breiter, G., Dinger, J.: The role of ontologies in autonomic computing systems. IBM Systems Journal. 43, 598–616 (2004).
[15] [17] J. Spring, “Monitoring Cloud Computing by Layer, Part 2”, Security & Privacy, IEEE 9(3), IEEE, 52–55, 2011.
[16] P. Hasselmeyer and N. d'Heureuse, “Towards holistic multi-tenant monitoring for virtual data centers”, NOMS 2010 IEEE/IFIP Network Operations and Management Symposium, Apr. 2010
[17] S. De Chaves, R. Uriarte, and C. Westphall. Toward an architecture for monitoring private clouds. Communications Magazine, IEEE, 49(12):130–137, Dec. 2011.